Supplemento alle dispense di Semantica

Luca Tesei

18 novembre 2002

Sommario

In queste note vengono introdotti gli array così come vengono trattati in Java e le principali operazioni su di essi. Di seguito vengono mostrati i tipici costrutti usati per manipolarli e alcuni semplici schemi di programmi per risolvere problemi di ricerca lineare. Vengono dati esempi di metodi che hanno più parametri e che restituiscono un valore. Infine, viene introdotto il meccanismo di ereditarietà delle classi.

1 Array

Finora abbiamo usato all'interno dei programmi le variabili di tipo `int`, `boolean` o riferimento ad oggetto. Può essere utile in certe applicazioni raggruppare un certo numero di variabili dello stesso tipo in un'unica variabile. Nella programmazione classica questo tipo nuovo di struttura si chiama *array* (in italiano si può tradurre con vettore). In matematica un vettore **v** (ad esempio di numeri reali) di dimensione n si può indicare con una scrittura del tipo $(v_0, v_1, \ldots v_{n-1})$ e il generico elemento si indica con v_i dove i è un *indice* che può assumere valori nell'insieme $\{0, 1, \ldots, n\}$.

Nei linguaggi di programmazione di solito si indica il generico elemento del vettore **v** con la scrittura `v[i]` dove `v` è una variabile di tipo array e `i` è una variabile di tipo `int` che deve avere un valore compreso tra `0` ed `N - 1` se `N` è la dimensione dichiarata dell'array. Il generico elemento `v[i]` è proprio una variabile di tipo T dove T è il tipo indicato nella dichiarazione dell'array. Come tutte le variabili, `v[i]` deve essere inizializzato, può essere assegnato con un valore di tipo T e può essere riferito all'interno di una espressione. In quest'ultimo caso il valore semantico è il valore che è presente in quel momento nella posizione `i` dell'array.

Nei linguaggi non ad oggetti come il Pascal o il C gli array sono variabili speciali che fanno parte dello stato allo stesso modo delle variabili "normali".

In Java invece gli array sono *oggetti*. Come tutti gli oggetti devono essere creati e sono raggiungibili attraverso variabili che contengono i riferimenti.

Introduciamo allora la sintassi per la dichiarazione di array ed estendiamo le categorie sintattiche dei comandi e delle espressioni per l'assegnamento e il riferimento di array:

```
Decl::=
    T[] Ide; | T Ide[];
    Ide[] Ide; | Ide Ide[];
    T[] Ide = new T[Exp]; | T Ide[] = new T[Exp];
    Ide[] Ide = new Ide[Exp]; | Ide Ide[] = new Ide[Exp];

Com::=
    Ide = new T[Exp]; |
    Ide = new Ide[Exp]; |
    Ide[Exp] = Exp; |
    Ide[Exp] = new Ide; |
    Ide[Exp].Ide = Exp; |
    Ide[Exp].Ide(Exp);

Exp::=
    Ide[Exp] |
    Ide[Exp].Ide

T::= int | boolean
```

Una dichiarazione del tipo `int[] a;` (oppure `boolean[] a;`) crea nel frame in testa alla pila di frame σ, che rappresenta la seconda componente dello stato corrente, una associazione per il nome `a` con il valore *null*. La variabile `a` può quindi contenere un riferimento ad un oggetto array di interi (oppure di booleani).

La dichiarazione `Foo[] a;` crea una variabile che può contenere un riferimento ad un oggetto array di variabili riferimento. Ogni elemento di un oggetto array puntato da `a` può quindi contenere un riferimento ad un oggetto della classe `Foo`.

Come per le variabili riferimento che conosciamo, contestualmente alla dichiarazione si può anche creare un oggetto: `int a = new int[3];` crea la variabile `a` nello stato corrente e crea un oggetto array nell'heap. Il riferimento al nuovo oggetto viene assegnato alla variabile `a`. L'array creato nell'heap contiene tre variabili intere inizializzate con $\overline{\omega}$. L'effetto della dichiarazione è visualizzato in Figura 1.

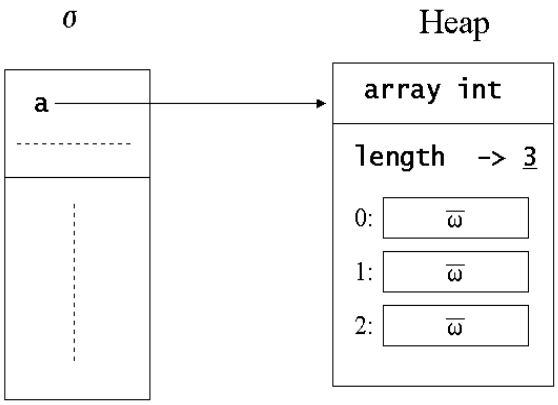

Figura 1: Effetto della dichiarazione `int[] a = new int[3];`

La dichiarazione `Foo fa = new Foo[4];` crea la variabile `a` nello stato corrente e crea un oggetto array nell'heap. Il riferimento al nuovo oggetto viene assegnato alla variabile `a`. L'array creato nell'heap contiene quattro variabili riferimento inizializzate con *null*. Queste variabili potranno contenere riferimenti ad oggetti della classe `Foo`. L'effetto della dichiarazione è visualizzato in Figura 2.

Come si vede dalla definizione della sintassi, il compilatore Java della Sun accetta due tipi di sintassi nella dichiarazione di array: è possibile porre le parentesi quadre vuote (`[]`) anche dopo il nome dell'identificatore che rappresenta il nome della variabile riferimento ad array che si sta creando. Ad esempio le dichiarazioni precedenti possono essere scritte anche come segue:

```
int a[];
boolean b[];
// oppure con inizializzazione
int a1[] = new int[3];

Foo fa[];
// oppure con inizializzazione
Foo fa1[] = new Foo[4];
```

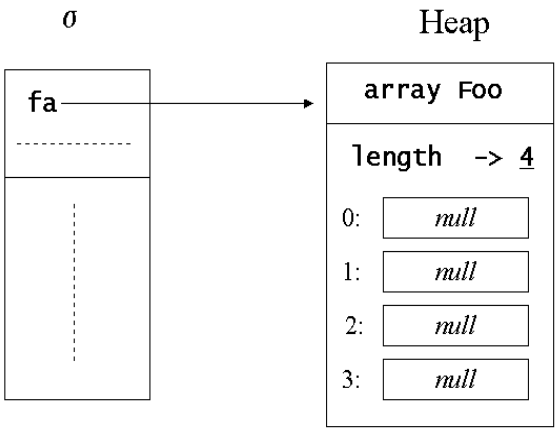

Figura 2: Effetto della dichiarazione `Foo[] fa = new Foo[4];`

Ogni oggetto array ha sempre un campo di nome `lenght` che contiene un intero positivo che è la dimensione dell'array, vale a dire il numero di elementi dello stesso. Quindi è sempre possibile valutare l'espressione `a.lenght` per sapere la dimensione dell'array puntato da `a`. È importante notare a questo punto che *la dimensione di un array viene specificata quando viene creato e non può più essere modificata*[1].

Quando un array è stato creato si possono assegnare i valori del tipo giusto agli elementi dell'array. Per far questo il comando da utilizzare è ovviamente l'assegnamento, ma a differenza dell'assegnamento di campi di oggetti normali, si usa la sintassi specifica: `a[0] = 3;` assegna il valore 3 al primo elemento dell'array puntato da `a`. A questo punto è importante notare che gli elementi di un qualsiasi array sono numerati da 0 ad $N-1$ dove N è la lunghezza dell'array specificata nel campo `lenght`. Ogni tentativo di leggere/scrivere un elemento al di fuori di questo intervallo provoca un errore dinamico (viene sollevata una eccezione). All'interno delle parentesi quadre può essere inserita una qualunque espressione che abbia un valore intero compreso nell'intervallo $[0, N-1]$. Consideriamo il seguente frammento di codice:

```
...
```

[1] L'implementazione di Java della Sun fornisce un tipo di oggetti chiamati `Vector` che sono array la cui dimensione può essere estesa dopo che questi sono stati creati

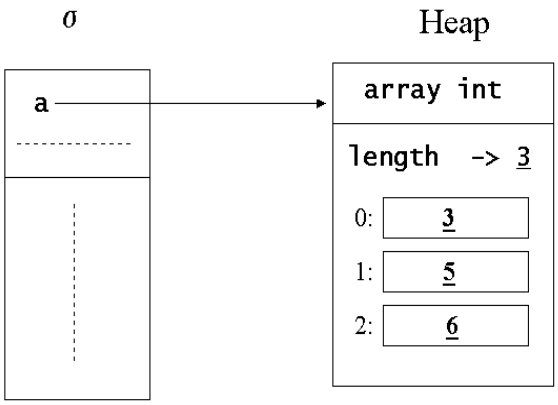

Figura 3: Effetto degli assegnamenti `(1)`, `(2)` e `(3)`

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

In Figura 3 è mostrata la situazione dopo l'esecuzione del frammento di codice. Come si vede si dichiara una variabile `i` che viene inizializzata a 0. L'assegnamento `(1)` inserisce il valore 3 nella posizione `i` dell'array puntato da `a`. Dato che il valore dell'espressione `i` è 0, sarà l'elemento in posizione 0 dell'array ad essere modificato. L'assegnamento `(2)` assegna il valore 5 alla posizione `i + 1` dell'array puntato da `a`. Il valore dell'espressione `i + 1` nello stato corrente è 1, quindi sarà l'elemento in posizione 1 dell'oggetto array ad essere modificato. L'assegnamento `(3)` assegna il valore 6 all'elemento in posizione 2 dell'array (cioè l'ultimo). Questo perché il valore dell'espressione `a.lenght - 1` nello stato corrente è 2 e l'espressione a destra dell"'=" vale 6 nello stato corrente. È importante notare a questo punto la differenza fra il campo `lenght` dell'oggetto array e gli elementi dell'array stesso. Il campo `length` viene considerato come un campo qualsiasi di un oggetto e viene quindi raggiunto tramite l'operatore ".". Invece gli elementi dell'oggetto array vengono raggiunti con l'operatore "`[ E ]`" dove `E` è una espressione intera.

Se un array non è di tipo `int` o `boolean`[2], allora i suoi elementi sono puntatori ad oggetti di una certa classe. Consideriamo l'array `fa` dichiarato sopra e supponiamo che la classe `Foo` sia definita come segue:

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

Possiamo usare gli elementi dell'array `fa` per mantenere dei riferimenti ad oggetti della classe `Foo`. Consideriamo il seguente frammento di codice:

```
...
fa[0] = new Foo;  (1)
fa[0].z = 15;     (2)
fa[0].b = true;   (3)
fa[1] = fa[0];    (4)
fa[2] = new Foo;  (5)
...
```

L'assegnamento `(1)` crea un oggetto della classe `Foo` e inserisce il suo riferimento nell'elemento 0 dell'array puntato da `fa`. I due successivi assegnamenti inizializzano i valori dei campi dell'oggetto appena creato. È da notare come i campi vengano raggiunti a partire dal riferimento contenuto in `fa[0]` e a cui correttamente viene applicato l'operatore ".". A sua volta il riferimento `fa[0]` viene ottenuto dal riferimento `fa` all'array e dall'operatore `[]`. In Figura 4 è mostrato lo stato dopo l'esecuzione dell'assegnamento `(3)`. L'assegnamento `(4)` opera a livello di riferimenti, nel senso che assegna all'elemento 1 dell'array `fa` il valore dell'espressione `fa[0]`. Questa espressione, se valutata nello stato corrente, restituisce il valore dell'elemento 0 dell'array `fa`, vale a dire il riferimento all'oggetto creato in `(1)`. L'effetto sarà quindi quello di avere due riferimenti a tale oggetto: uno è quello che già avevamo e l'altro si troverà nell'elemento 1 dell'array. L'assegnamento `(5)` ha l'effetto di creare un nuovo oggetto della classe `Foo` e di inserire il suo riferimento nell'elemento 2 dell'array. Il risultato è mostrato in Figura 5.

[2]Ci limitiamo a considerare solo questi due tipi base, mentre nel Java completo le stesse considerazioni si applicano a tutti i tipi base: `float, double, byte, short,` etc.

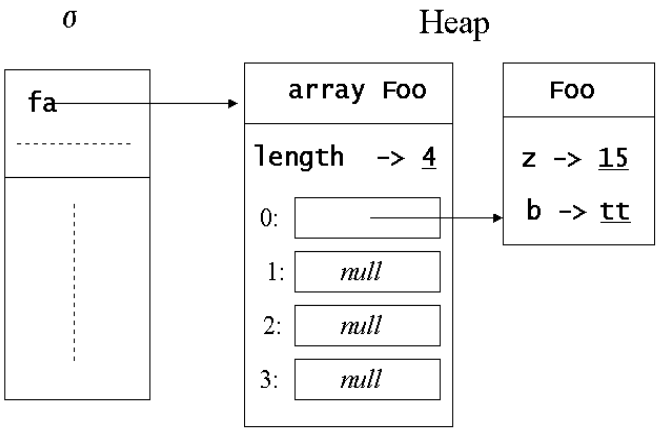

Figura 4: Effetto degli assegnamenti `(1)`, `(2)` e `(3)`

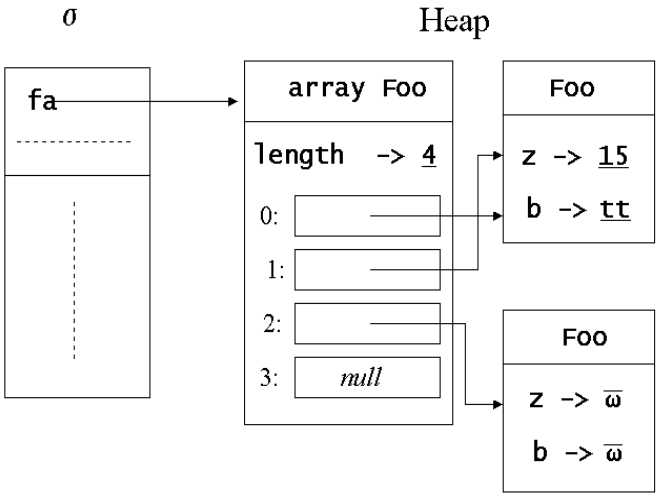

Figura 5: Effetto degli assegnamenti `(4)` e `(5)`

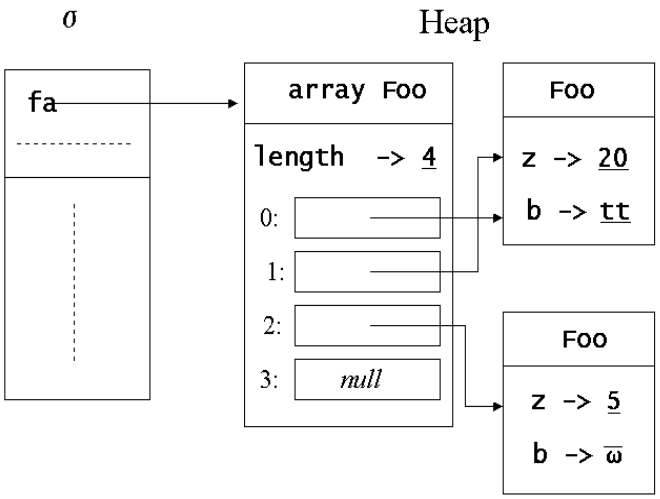

Figura 6: Effetto degli assegnamenti `(6)` e `(7)`

Vediamo a questo punto come si chiamano i metodi degli oggetti della classe `Foo` tramite i riferimenti che abbiamo nell'array. Consideriamo i seguenti comandi a partire dallo stato rappresentato in Figura 5:

```
fa[2].z = 5;                (6)
fa[1].cond_inc(fa[2].z); (7)
```

L'assegnamento `(6)` assegna il campo `z` del secondo oggetto creato con il valore 5. Il comando (6) è la chiamata del metodo `cond_inc` dell'oggetto riferito dall'elemento 1 dell'array. Tale oggetto, come risulta dallo stato, è il primo oggetto che è stato creato e che è riferito anche dal primo elemento dell'array. Il metodo incrementa il campo `z` dell'oggetto con il valore passato nella chiamata solo se il campo `b` dell'oggetto ha valore vero. Il valore passato al metodo è il valore dell'espressione `fa[2].z`, cioè il valore del campo `z` del secondo oggetto creato. Tale valore è 5. Il risultato è mostrato in Figura 6.

2 Costrutti tipici nell'uso di array e schemi di programmi

In questa sezione vediamo alcuni esempi di uso di array e due schemi di programmi che possono venire applicati per cercare il primo elemento di un array che soddisfa una certa proprietà.

2.1 Il comando `for`

Uno dei vantaggi dell'uso degli array è la possibilità di scrivere concisamente del codice che deve eseguire alcune operazioni su tutto un insieme di variabili dello stesso tipo. Questo viene fatto usando gli indici e un comando particolare: il ciclo `for` la cui sintassi è la seguente:

```
Com ::= for (Com_or_Decl Exp; Com) Com

Com_or_Decl ::= Com | Decl
```

Il comando `for (C1 E; C2) C` consiste nell'eseguire il comando (o la dichiarazione) `C1` (detto anche inizializzazione) in un nuovo blocco e, a partire dal nuovo stato ottenuto, eseguire il ciclo `while (E) { C C2 }`. Le regole di semantica operazionale sono le seguenti:

$$com_{for-com} \quad \frac{\texttt{C1} \in \texttt{Com} \quad \langle \texttt{C1}, \langle \rho_c, \omega.\sigma, \zeta \rangle \rangle \rightarrow_{com} \langle \sigma', \zeta' \rangle \quad \langle \texttt{while (E) \{C C2\}}, \langle \rho_c, \sigma', \zeta' \rangle \rangle \rightarrow_{com} \langle \phi.\sigma'', \zeta'' \rangle}{\langle \texttt{for (C1 E; C2) C}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{com} \langle \sigma'', \zeta'' \rangle}$$

$$com_{for-dec} \quad \frac{\texttt{C1} \in \texttt{Dec} \quad \langle \texttt{C1}, \langle \rho_c, \omega.\sigma, \zeta \rangle \rangle \rightarrow_{dec} \langle \sigma', \zeta' \rangle \quad \langle \texttt{while (E) \{C C2\}}, \langle \rho_c, \sigma', \zeta' \rangle \rangle \rightarrow_{com} \langle \phi.\sigma'', \zeta'' \rangle}{\langle \texttt{for (C1 E; C2) C}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{com} \langle \sigma'', \zeta'' \rangle}$$

Nella pratica il comando `for` viene usato quasi sempre associato ad operazioni su array. Vediamo un tipico esempio di uso:

```
int[] a = new int[10];
for (int i = 0; i < a.length; i = i + 1)
   a[i] = i;
```

Innanzitutto facciamo una osservazione sintattica. Il comando `i = i + 1` non è seguito dal punto e virgola. Questo è lecito, ma non è in accordo con la sintassi che abbiamo dato tramite le produzioni viste sopra. In realtà in Java in quel punto non è richiesto il punto e virgola poichè c'è la parentesi tonda che chiude il comando. Si lascia al lettore la modifica della sintassi in modo da riflettere questo fatto.

La prima riga del frammento di codice dichiara una variabile riferimento ad un array di interi e crea l'oggetto array. Lo scopo del ciclo `for` in questo caso è quello di inizializzare tutti gli elementi dell'array con un valore pari alla propria posizione nell'array. Il comando di inizializzazione è una dichiarazione di una variabile intera `i` che viene inizializzata a 0. Si noti che, per

come è definita la semantica, tale variabile esiste solo all'interno del blocco creato appositamente per eseguire il `for`. Essa scompare dopo l'esecuzione del ciclo poiché il frame che abbiamo creato per eseguire il comando viene rimosso dalla testa della pila dopo l'uscita. La variabile `i` servirà da indice per eseguire l'operazione di inizializzazione di tutti gli elementi dell'array. La condizione del ciclo infatti diventerà falsa non appena `i` assumerà il valore `a.length`. Il comando `C2` in questo caso è l'incremento dell'indice. Questo viene eseguito ad ogni iterazione del ciclo *dopo* che è stato eseguito il comando `C`, che in questo caso particolare è l'assegnamento all'`i`-esimo elemento dell'array del valore `i`.

Seguiamo passo passo cosa avviene. Immediatamente prima del `for` si ha che $\forall i \in [0, \texttt{a.lenght} - 1].\texttt{a}[\texttt{i}] = \overline{\omega}$. Dopo l'inizializzazione di `i` il ciclo parte poiché ovviamente $\texttt{i} = 0 < \texttt{a.lenght} = 10$. Viene quindi eseguito il comando `a[i]=i;` in uno stato in cui `i`=0. L'effetto sarà quello di assegnare il valore 0 all'elemento 0 dell'array. A questo punto la semantica prescrive di eseguire il comando `C2`, cioè `i = i + 1;`. L'effetto sarà quello di arrivare in uno stato in cui `i`=1. A questo punto bisogna rivalutare l'espressione `E` in questo stato. Essa sarà ancora vera e quindi si farà un'altra iterazione. La seconda iterazione quindi consiste nell'eseguire `a[i]=i;` in uno stato in cui `i`=1. L'effetto sarà quello di assegnare il valore 1 all'elemento 1 dell'array. Si prosegue eseguendo di nuovo `i = i + 1;` come prescritto dalla semantica. Si giunge pertanto in uno stato in cui `i`=2 e si ricomincia rivalutando l'espressione in questo nuovo stato. A questo punto è facile convincersi che il tutto andrà avanti fino all'iterazione in cui viene eseguito `a[i] = i;` in uno stato in cui `i`=9. Dopo questo assegnamento avremo completato l'inizializzazione di tutti gli elementi dell'array. A questo punto il comando `i = i + 1;` porterà in uno stato in cui `i`=10 in cui la guardia è falsa. All'uscita del `for`, quindi, la variabile `i` scompare e si avrà che $\forall \texttt{i} \in [0, \texttt{a.lenght} - 1].\texttt{a}[\texttt{i}] = \texttt{i}$, come volevamo.

Vediamo un altro esempio. Supponiamo di avere un array di interi già inizializzato e riferito dalla variabile `a`. Vogliamo scrivere un comando che ponga a zero l'ultimo elemento dell'array e sposti tutti gli elementi di una posizione verso sinistra. Il primo valore dell'array viene perduto:

```
int i;
for (i = 0; i < a.lenght - 1; i = i + 1)
   a[i] = a[i+1];
a[i] = 0;
```

In questo caso la variabile `i` non è dichiarata nel blocco del `for` quindi essa rimarrà anche dopo l'esecuzione del ciclo e manterrà il suo valore. Il ciclo

viene eseguito per tutti gli elementi dell'array tranne l'ultimo (l'elemento in posizione `a.lenght - 1`). Ad ogni iterazione l'elemento in posizione `i + 1` viene copiato nella posizione `i`. All'uscita del ciclo la variabile `i` varrà esattamente `a.lenght - 1` e quindi viene riusata per assegnare zero all'ultimo elemento dell'array, come volevamo.

Spesso si usano anche cicli `for` annidati. Un esempio classico è quello dell'algoritmo di ordinamento denominato *bubble-sort*. Supponiamo di avere un array di interi già inizializzato e riferito dalla variabile `a`. L'algoritmo di ordinamento ordina i valori effettuando degli scambi: se viene trovato in una posizione `i` un elemento che è maggiore di un altro che si trova in una posizione `j`, maggiore di `i`, allora i due vengono scambiati di posto. Se lo si fa per tutti i valori possibili di `i` e `j` alla fine si otterrà un array di elementi ordinato, vale a dire un array in cui $\forall \texttt{i}, \texttt{j} \in [0, \texttt{a.lenght} - 1].\texttt{i} \leq \texttt{j} \Rightarrow \texttt{a}[\texttt{i}] \leq \texttt{a}[\texttt{j}]$.

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

Il ciclo più esterno gestisce l'incremento dell'indice `i` che arriverà alla penultima posizione dell'array. Per ogni posizione `i`, si esegue un ciclo a partire dalla posizione successiva (`i + 1`) fino alla fine dell'array. Questo ciclo interno gestisce l'indice `j`. Per ogni posizione `j` maggiore di `i` si controlla se bisogna spostare l'elemento in posizione `i` (cioè se questo è maggiore di quello in posizione `j`). Per effettuare lo scambio bisogna usare una variabile di appoggio (`buf`) per mantenere il valore di `a[i]`. Una volta salvato quest'ultimo in `buf` la posizione `i` dell'array viene assegnata con il valore nella posizione `j` ed infine il valore salvato in `buf` viene posto in posizione `j`.

I marcatori `//` stanno a indicare commenti nel codice. Essi fanno in modo che i caratteri che li seguono, fino alla fine della linea, vengano ignorati dal compilatore.

2.2 Ricerca lineare

Vediamo ora due schemi di programmi molto semplici che possono essere applicati ogni volta che si deve cercare, in un array, la posizione del primo elemento che soddisfa una certa proprietà. Tale proprietà è definita sui valori del tipo dell'array e di solito viene scritta come espressione booleana. Sia allora $P(v)$ una certa proprietà definita sui valori contenuti negli elementi di un array riferito da una variabile `a`. Il primo schema che consideriamo si chiama *ricerca lineare certa*. Esso si può applicare sotto l'ipotesi che esista almeno un elemento dell'array che soddisfa la proprietà P. Per trovare il primo elemento che soddisfa P si può allora scrivere il seguente codice:

```
int i = 0;
while (!P(a[i]))
  i = i + 1;
(1)
```

Nel punto `(1)` del programma si ha allora che la posizione `i` è quella in cui si trova il primo elemento dell'array che soddisfa P. Vediamo un esempio. Supponiamo di avere un array a valori interi e supponiamo che esista almeno un elemento pari. La nostra proprietà P sarà allora $P(n) : n \ \% \ 2 == 0$. Istanziando lo schema avremo il seguente codice:

```
int i = 0;
while (a[i] % 2 != 0)
  i = i + 1;
```

All'uscita del `while` avremo che la variabile `i` contiene la posizione del primo elemento pari dell'array.

Se non si ha la certezza che l'array contenga almeno un elemento che soddisfa la proprietà P, allora bisogna applicare un altro schema denominato ricerca lineare incerta. Esso troverà, se esiste, il primo elemento dell'array che soddisfa P oppure, se non esiste nessun elemento dell'array che soddisfa P, esso conclude arrivando in uno stato in cui una variabile booleana ha valore falso:

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

Nel punto del programma `(1)`, se la variabile booleana `trovato` vale **tt** allora il primo elemento dell'array che soddisfa P si trova nella posizione contenuta nella variabile `i`. Se la variabile `trovato` vale **ff** allora si ha che nessun elemento dell'array soddisfa P.

Vediamo un esempio. Supponiamo di avere un array di interi e cerchiamo il primo elemento che soddisfa $P(n) : n > 100$.

```
int i = 0;
boolean trovato = false;
while (i < a.length && !trovato)
   if (a[i] > 100)
     trovato = true;
   else
     i = i + 1;
```

Questi schemi possono essere applicati anche se non si deve operare su un array, oppure alcune semplici varianti di essi possono essere applicate per verificare una proprietà più complessa. Un esempio di applicazione della ricerca lineare certa è quello in cui si cerca, in un intervallo $[0, N]$ di numeri naturali, la parte intera della radice quadrata di N. È chiaro che questo numero si trova necessariamente nell'intervallo $[0, N]$:

```
int i = 0;
while ((i+1)*(i+1) <= N)
  i = i + 1;
```

All'uscita del ciclo la variabile `i` contiene la parte intera della radice quadrata di N.

Una variante della ricerca lineare incerta può essere quella in cui si verifica se un certo array di interi è ordinato. In questo caso è sufficiente verificare che ogni elemento dell'array, tranne l'ultimo, è minore o uguale del successivo. Quindi possiamo applicare la ricerca lineare incerta sull'array (trascurando l'ultimo elemento) e cercare se c'è un elemento che è maggiore del successivo. Se un tale elemento non è presente nell'intervallo $[0, \texttt{a.length} - 2]$, allora abbiamo verificato che l'array è ordinato:

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

All'uscita del ciclo l'array è ordinato se e solo se la variabile booleana `trovato` vale **ff**.

3 Metodi a più parametri e che restituiscono un valore

Nel frammento di Java che abbiamo esaminato abbiamo fatto alcune semplificazioni. Una di queste è quella che forza tutti i metodi delle classi a restituire il tipo `void` (il tipo vuoto e quindi nessun valore restituito) e ad avere solo un parametro. Chiaramente nelle applicazioni vere proprie queste limitazioni sono troppo restrittive. Infatti nel linguaggio Java un metodo può avere un numero qualsiasi di parametri (anche nessun parametro). Inoltre esso può restituire un valore. In quest'ultimo caso si ha allora che la chiamata di un metodo è considerata come un'espressione. La sintassi che permette ad un metodo di avere più parametri è la seguente:

```
Method_decl ::=
   public void Ide ( Param_List ) Block |
   public T Ide ( Param_List ) Block |
   public Ide Ide (Param_List ) Block
```

Nel secondo caso il metodo restituisce un tipo base, mentre nel terzo caso il metodo `public c m ( PL ) B` restituisce un riferimento ad un oggetto della classe `c`. La lista di parametri (separati da una virgola) abbiamo detto che può essere composta da zero o più parametri:

```
Param_List ::=
   Empty_Param_List |
   Non_Empty_Param_List

Empty_Param_List ::= <epsilon>

Non_Empty_Param_List ::=
   Param, Param_List |
   Param

Param ::=
   T Ide |    (1)
   Ide Ide |  (2)
   T[] Ide |  (3)
   Ide[] Ide  (4)
```

Nel caso `(1)` il parametro è un valore di tipo base (intero, booleano eccetera). Nel caso `(2)` il parametro `c y` è un riferimento ad un oggetto della classe `c`. Nel caso `(3)` il parametro è un riferimento ad un array di elementi di tipo base e nel caso `(4)` il parametro `c[] a` è un riferimento ad un array i cui elementi sono riferimenti ad oggetti della classe `c`.

La lista dei parametri di un metodo costituisce la *segnatura* del qmetodo e in effetti è tutto ciò che bisogna sapere per poter chiamare un metodo (oltre ovviamente ad avere una descrizione di ciò che fa il metodo stesso) senza alcun bisogno di conoscerne l'implementazione, cioè il blocco di comandi che esegue. Molte delle classi fornite dall'implementazione del linguaggio Java della Sun possono essere usate semplicemente conoscendo solo la segnatura dei vari metodi e la descrizione di ciò che il metodo fa se i valori dei parametri attuali soddisfano certe proprietà.

A questo punto diamo la nuova sintassi della chiamata di metodi che restituiscono il tipo `void`:

```
Com ::= Ide.Ide( Expr_List )

Expr_List ::=
    Empty_Expr_List |
    Non_Empty_Expr_List

Empty_Expr_List ::= <epsilon>

Non_Empty_Expr_List ::=
    Exp, Expr_List |
    Exp
```

I metodi che invece restituiscono dei valori sono considerati espressioni, ma semanticamente la loro valutazione può modificare lo stato e quindi le espressioni di questo tipo non possono essere valutate con il sistema $\rightarrow_{exp}$ che abbiamo visto. Questo sistema deve essere ridefinito in modo tale che la valutazione di una espressione restituisca un coppia formata dal valore dell'espressione e da uno heap. Questo perchè una espressione potrebbe contenere una sottoespressione che è la chiamata di un metodo e, come sappiamo, un metodo può, in generale, modificare l'heap. La ridefinizione del sistema di valutazione delle espressioni è lasciata al lettore. Vediamo la sintassi:

```
Exp ::= Ide.Ide( Expr_List )
```

I metodi restituiscono un valore e terminano la loro esecuzione tramite il comando `return`:

```
Com::= return Exp;
```

L'esecuzione di `return E;` all'interno di un metodo consiste nel valutare l'espressione E nello stato corrente e nel terminare l'esecuzione del metodo restituendo come risultato il valore calcolato e l'heap corrente.

Vediamo un esempio. Scriviamo un metodo che, presi come parametri un riferimento ad un array di interi ed un intero restituisce un valore booleano che indica se qualche elemento dell'array contiene il valore intero che abbiamo passato come secondo parametro.

```
/** Cerca un valore intero in un array di interi
  *
  * @param a: un riferimento ad un array di interi
  * @param x: il valore intero da ricercare nell'array
  *
  * @return vero se il valore e' presente nell'array,
  *         falso altrimenti
  */
public boolean occurs_in(int[] a, int x) {
  // Applico lo schema della ricerca lineare incerta
  int i = 0;
  boolean trovato = false;
  while (i < a.length && !trovato)
    if (a[i]==x)
      trovato = true;
    else
      i = i + 1;
  return trovato;
}
```

Dopo il ciclo il metodo restituisce il valore della variabile booleana `trovato`.

Il commento fra i marcatori `/**` e `*/` è un tipo speciale di commento che può essere posto prima della dichiarazione di una classe o di un metodo. All'interno si possono utilizzare i marcatori `@param` e `@return` per descrivere le ipotesi che devono soddisfare i parametri quando si chiama il metodo e il significato del valore restituito, rispettivamente. Eseguendo poi l'utility `javadoc` fornita con la Jdk della Sun è possibile ottenere automaticamente una documentazione del programma sotto forma di pagine html.

Vediamo un altro esempio. Consideriamo i punti del piano cartesiano a coordinate intere. Possiamo rappresentarli come oggetti. Lo stato conterrà i valori delle coordinate cartesiane. Definiamo tre metodi: uno per

determinare se un certo punto si trova sull'origine, un altro per resettare i valori delle coordinate e l'ultimo per sommare altre coordinate alle coordinate dell'oggetto.

Poi definiamo una classe `Set_of_Points` i cui oggetti rappresentano insiemi di punti. Rappresentiamo un insieme di punti con un array di oggetti punto. Definiamo un metodo che crea un punto con le coordinare del punto medio dell'insieme dei punti di un oggetto di tipo `Set_of_Points`.

Scriviamo un semplice blocco, infine, per usare le funzionalità delle nostre due classi.

```
prog {
  /** Rappresenta un punto di coordinate intere nel
    * piano cartesiano
    */
  class Point {
     public int x; //ascissa
     public int y; //ordinata

     /** Determina se l'oggetto rappresenta un punto
       * che si trova sull'origine
       */
     public boolean is_origin() {
        if (this.x==0 && this.y==0)
          return true;
        else
          return false;
     }

     /** Resetta le coordinate del punto a (0,0)
       */
     public void clear() {
        this.x = 0;
        this.y = 0;
     }

     /** Sposta il punto rappresentato dall'oggetto
       * sommandogli un altro punto di coordinate date
       * @param x: l'ascissa da sommare
       * @param y: l'ordinata da sommare
       */
     public void somma( int x, int y ) {
```

```
      this.x = this.x + x;
      this.y = this.y + y;
   }
}


/** Un oggetto di questa classe rappresenta un insieme
  * di punti a coordinate intere nel piano cartesiano
  */
class Set_of_Points {
   public Point[] set; // l'insieme lo rappresento con un array

   /** Calcola le coordinate del punto medio dei punti che si
     * trovano nell'array set dell'oggetto
     *
     * @return un nuovo oggetto Point con le coordinate del punto
     *         medio
     */
   public Point midpoint() {
     int sommax = 0; // accumulatore per la somma delle ascisse
     int sommay = 0; // accumulatore per la somma delle ordinate
     for (int i = 0; i < this.set.length; i = i + 1) {
        sommax = sommax + this.set[i].x;
        sommay = sommay + this.set[i].y;
     }
     Point medio = new Point;
     medio.x = sommax / this.set.length; // media delle ascisse
     medio.y = sommay / this.set.lenght; // media delle ordinate
     return medio;
   }
 }
{
 Set_of_Points s = new Set_of_Points; // creo l'oggetto della classe
 s.set = new Point[3];  // creo l'array che contiene i punti
 // creo i tre punti e li inizializzo
 s.set[0] = new Point;
 s.set[0].x = -3;
 s.set[0].y = 50;
 s.set[1] = new Point;
 s.set[1].x = 0;
 s.set[1].y = 4;
```

```
    s.set[2] = new Point;
    s.set[2].x = -15;
    s.set[2].y = -30;
    // calcolo il punto medio
    Point m = s.midpoint();
    // l'oggetto riferito da m e' stato creato all'interno
    // del metodo e ha il seguente stato:
    // m.x == -6
    // m.y == 8
    // Sommo al punto medio le coordinate (4,4) se non sta nell'origine
    if (!m.is_origin())
      m.somma(4,4);
    // Il nuovo stato di m e' il seguente:
    // m.x == -2
    // m.y == 12
  }
}
```

Nel metodo `midpoint()` l'operatore "." viene applicato ripetutamente in una stessa espressione. Ciò non è previsto dalla nostra sintassi, ma è ovviamente corretto. Infatti l'espressione `this.set[i].x`, ad esempio, è corretta poiché `this` e `set[i]` sono entrambi riferimenti ad oggetto. Si lascia al lettore il compito di modificare la sintassi in modo da contemplare casi sintattici di questo tipo, vale a dire applicazioni multiple (in effetti un qualunque numero) degli operatori "." e `[]`[3] nella stessa espressione, chiamata di metodo o assegnamento.

Vediamo infine come devono essere scritti i programmi che abbiamo visto per poter essere compilati ed eseguiti nella reale implementazione di Java della Sun. Innanzitutto le classi devono essere pubbliche: ciò si ottiene anteponendo alla parola `class` l'attributo `public`. Poi ogni classe deve essere scritta in un file apposito che ha lo stesso nome della classe e l'estensione `.java`. Il blocco di comandi del nostro programma deve figurare come metodo statico e pubblico di una classe (noi scegliamo `Set_of_Points`) e tale metodo deve chiamarsi `main` ed avere come parametro un array di stringhe che rappresentano i valori passati al programma sulla linea di comando. Inoltre, quando si crea un nuovo oggetto con l'operatore `new`, il nome della classe dopo `new` deve essere seguito da una parentesi tonda aperta e una chiusa. Tale nome è il nome di un metodo speciale che non prende nessun parametro e che è definito implicitamente per ogni classe. Il metodo in questione ha lo

[3] L'applicazione ripetuta dell'operatore `[]` porta alla definizione di array multidimensionali (matrici)

stesso nome della classe e si chiama *costruttore*. Di default non fa assolutamente nulla (oltre alla creazione fisica dell'oggetto nello heap, cioè quello che fa il nostro sottosistema $\rightarrow_{new}$). Tale metodo può essere ridefinito in modo da inizializzare alcuni campi dell'oggetto, oppure può essere definito un metodo con lo stesso nome che ha alcuni parametri. In questo caso il metodo inizializzerà i campi utilizzando i valori forniti dai parametri. Vediamo il codice:

```
/**
 * file Point.java
 * @version 1.0 18/11/2002
 * @author Luca Tesei
 *
 * Rappresenta un punto di coordinate intere nel
 * piano cartesiano
 */
public class Point {
    /** Ascissa */
    public int x;
    /** Ordinata */
    public int y;

    /** Determina se l'oggetto rappresenta un punto
     * che si trova sull'origine
     */
    public boolean is_origin() {
        if (this.x==0 && this.y==0)
    return true;
        else
    return false;
    }

    /** Resetta le coordinate del punto a (0,0)
     */
    public void clear() {
        this.x = 0;
        this.y = 0;
    }

    /** Sposta il punto rappresentato dall'oggetto
     * sommandogli un altro punto di coordinate date
```

```
     * @param x l'ascissa da sommare
     * @param y l'ordinata da sommare
     */
    public void somma( int x, int y ) {
        this.x = this.x + x;
        this.y = this.y + y;
    }
}

/** file Set_of_Points.java
 * @version 1.0 18/11/2002
 * @author Luca Tesei
 *
 * Un oggetto di questa classe rappresenta un insieme
 * di punti a coordinate intere nel piano cartesiano
 */
public class Set_of_Points {
    /** L'array che rappresenta l'insieme di punti
     */
    public Point[] set;

    /** Calcola le coordinate del punto medio dei punti che si
     * trovano nell'array set dell'oggetto
     *
     * @return un nuovo oggetto Point con le coordinate del punto
     *         medio
     */
    public Point midpoint() {
int sommax = 0; // accumulatore per la somma delle ascisse
int sommay = 0; // accumulatore per la somma delle ordinate
for (int i = 0; i < this.set.length; i = i + 1) {
    sommax = sommax + this.set[i].x;
    sommay = sommay + this.set[i].y;
}
Point medio = new Point();
medio.x = sommax / this.set.length; // media delle ascisse
medio.y = sommay / this.set.length; // media delle ordinate
return medio;
    }
    static public void main(String argv[]) {
Set_of_Points s = new Set_of_Points(); // creo l'oggetto della classe
```

```
s.set = new Point[3];  // creo l'array che contiene i punti
// creo i tre punti e li inizializzo
s.set[0] = new Point();
s.set[0].x = -3;
s.set[0].y = 50;
s.set[1] = new Point();
s.set[1].x = 0;
s.set[1].y = 4;
s.set[2] = new Point();
s.set[2].x = -15;
s.set[2].y = -30;
// calcolo il punto medio
Point m = s.midpoint();
// l'oggetto riferito da m e' stato creato all'interno
// del metodo e ha il seguente stato:
// m.x == -6
// m.y == 8
System.out.println ("m.x == " + m.x);
System.out.println ("m.y == " + m.y);
// Sommo al punto medio le coordinate (4,4) se non sta nell'origine
if (!m.is_origin())
    m.somma(4,4);
// Il nuovo stato di m e' il seguente:
// m.x == -2
// m.y == 12
System.out.println ("m.x == " + m.x);
System.out.println ("m.y == " + m.y);
    }
}
```

Se i sorgenti di cui sopra vengono compilati e si lancia il main si ha la seguente videata:

```
C:\Luca\java> javac Point.java

C:\Luca\java> javac Set_of_Points.java

C:\Luca\java> java Set_of_Points
m.x == -6
m.y == 8
m.x == -2
```

```
m.y == 12
C:\Luca\java>
```

4 Ereditarietà

Uno dei vantaggi offerti dalla programmazione ad oggetti è quello di poter *estendere* una classe o, usando un'altra espressione per lo stesso meccanismo, quello di poter *costruire sottoclassi* di una certa classe già definita. Le estensioni possono allora essere usate come vere e proprie classi.

In Java è possibile per una classe ereditare da una ed una sola classe (che viene detta *superclasse*[4]. La classe che eredita viene detta *sottoclasse*. La sottoclasse eredita tutti i campi e i metodi della superclasse e può aggiungere nuovi metodi o campi. La sottoclasse può anche ridefinire i metodi o i campi della superclasse seguendo però regole precise.

Come esempio possiamo considerare la classe `Point` dei punti del piano cartesiano a coordinate intere che abbiamo definito nella sezione precedente. Possiamo creare una nuova classe `Pixel` dei pixel di uno schermo usando tutto ciò che già abbiamo scritto per `Point`. Il punto fondamentale è che un pixel ha due coordiante cartesiane intere. Oltre a queste ha un altro attributo che è il colore. Nel linguaggio delle classi questo si traduce dicendo che la classe `Pixel` estende la classe `Point` aggiungendo un campo `color`. La classe `Pixel` eredita i campi `x` e `y` della classe `Point` e tutti i metodi che abbiamo definito per i punti. Tuttavia il metodo `clear()` della classe `Point` non si comporterebbe in maniera giusta su un pixel perchè il pixel ha anche il campo `color` che non viene resettato dal metodo `clear()` dei punti. Quindi possiamo pensare di ridefinire il metodo `clear()` nella classe `Pixel` facendogli resettare anche il colore:

```
/**
 * File Pixel.java
 * @version 1.0 18/11/2002
 * @author Luca Tesei
 *
 * Questa classe rappresenta il pixel di uno schermo: ha due
 * coordinate come Point e tre interi che rappresentano il colore
 */
public class Pixel extends Point {
```

[4]In Java è possibile l'ereditarietà multipla (cioè da più superclassi) solo se le superclassi in questione sono *interfacce* o classi astratte. In questo caso si dice che la sottoclasse *implementa* le interfacce/superclassi astratte

```
    /** Rappresento il colore con la tripla di valori per il rosso,
     * verde e blu che sono i colori con cui si formano gli altri negli
     * schermi RGB
     */
    int red;
    int green;
    int blue;

    /**
     * Ridefinisco il metodo clear per resettare a nero il colore
     */
   public void clear() {
      super.clear(); // chiamo il metodo clear della superclasse
      // completo il reset ponendo a zero i valori dei tre colori
      this.red = 0;
      this.green = 0;
      this.blue = 0;
   }
// per il metodo main (nello stesso file) vedi sotto
```

Come si vede viene scritto solo quello che c'è di nuovo rispetto alla superclasse, in questo caso i nuovi campi per il colore, e quello che viene ridefinito, in questo caso il metodo `clear()`. Si noti che all'interno della ridefinizione di questo metodo abbiamo usato la parola `super` come riferimento. Nei metodi di una sottoclasse è possibile invocare i metodi della supercalsse utilizzando la parola `super` che può essere pensata come un riferimento all'oggetto in questione, ma visto come oggetto della superclasse. Nel nostro esempio la chiamata `super.clear()` fa in modo che venga eseguito il codice del metodo `clear()` della classe `Point` che resetta i campi `x` e `y`. In seguito vengono resettati i campi che rappresentano il colore.

Una caratteristica importante dell'ereditarietà è il fatto che gli oggetti di una sottoclasse possono essere utilizzati dal codice scritto per la superclasse. Nel nostro esempio possiamo usare gli oggetti della classe `Pixel` nel codice scritto per oggetti di tipo `Point`. Questa caratteristica si chiama *polimorfismo*: un oggetto di tipo `Pixel` può avere molte (*poli*) forme (*morfismo*) e può essere utilizzato sia come oggetto di tipo `Pixel` sia come oggetto di tipo `Point`. Per illustrare questa caratteristica utilizziamo la classe `Set_of_Points` definita nella sezione precedente e creiamo un insieme di punti che sono in realtà oggetti pixel. Vedremo come con il polimorfismo possiamo utilizzare ancora il metodo `midpoint()` per calcolare il punto medio

anche tra pixel. Supponiamo di avere definito le classi `Point`, `Set_of_Points` e `Pixel` come sopra e consideriamo il seguente blocco di comandi (main):

```
// seguito del file Pixel.java
public static void main(String argv[]) {
  Set_of_Points s = new Set_of_Points();
  s.set = new Point[2]; // creo l'array di punti
  s.set[0] = new Pixel();
  /* ho creato un oggetto di tipo Pixel e con il polimorfismo
   * ho pututo farlo puntare da una variabile (l'elemento dell'array)
   * che riferisce oggetti di tipo Point.
   */
  // Assegno al pixel, visto come Point, le coordinate (20,10)
  s.set[0].x = 20;
  s.set[0].y = 10;
  // Creo un oggetto Pixel e gli assegno il colore rosso, oltre alle
  // coordinate (10,20)
  Pixel p = new Pixel();
  p.x = 10;
  p.y = 20;
  p.red = 255;
  p.green = 0;
  p.blue = 0;
  // Assegno al secondo elemento dell'array il riferimento al nuovo
  // Pixel creato
  s.set[1] = p;
  //A questo punto posso chiamare il metodo per calcolare il punto medio
  Point p1;
  p1 = s.midpoint();
  // p1.x == 15 e p1.y == 15
  System.out.println("p1.x == " + p1.x);
  System.out.println("p1.y == " + p1.y);
  //resetto le coordinate e il colore del secondo oggetto creato
  p.clear();
}
}
```

Se si compila il file e si lancia il programma si ottiene la seguente videata:

```
C:\Luca\java> javac Pixel.java
```

```
C:\Luca\java> java Pixel
p1.x == 15
p1.y == 15
C:\Luca\java>
```

A questo punto è possibile creare la documentazione in formato `html` delle nostre classi utilizzando l'utility apposita fornita con la sdk:

```
C:\Luca\java> javadoc Point.java Set_of_Points.java Pixel.java
Loading source file Point.java...
Loading source file Set_of_Points.java...
Loading source file Pixel.java...
Constructing Javadoc information...
Standard Doclet version 1.4.0

Generating constant-values.html...
Building tree for all the packages and classes...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
Generating deprecated-list.html...
Building index for all classes...
Generating allclasses-frame.html...
Generating allclasses-noframe.html...
Generating index.html...
Generating packages.html...
Generating Pixel.html...
Generating Point.html...
Generating Set_of_Points.html...
Generating package-list...
Generating help-doc.html...
Generating stylesheet.css...

C:\Luca\java>
```

Il file `index.html` può essere aperto con un browser e vi si trova la documentazione delle nostre classi nel formato classico della documentazione Java.