Introduzione all'ambiente

MATLAB©

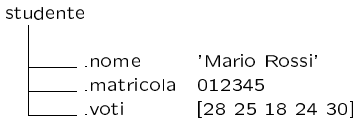
Definizione e manipolazione di strutture.

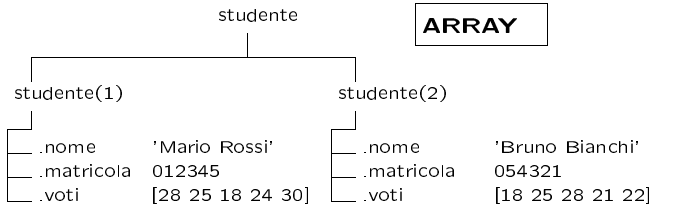
Classi ed oggetti.

Lezione 5 *Paola Gervasio*

Struttura è un array con dei "contenitori di dati" chiamati *campi (=fields)*. Ogni campo ha un nome e può contenere dati di un tipo diverso dal tipo degli altri campi.

Es.:





Come generare la struttura "studente" in MATLAB:
1^a possibilità:
Ogni campo è definito con `nome_struttura.nome_campo=....`

```
>> studente.nome='Mario Rossi';
>> studente.matricola=012345;
>> studente.voti=[28 25 18 24 30];
>> studente
studente =
         nome: 'Mario Rossi'
    matricola: 12345
         voti: [28 25 18 24 30]
```

Per aggiungere dati relativi ad altri studenti:

```
>> studente(2).nome='Bruno Bianchi';
>> studente(2).matricola=054321;
>> studente(2).voti=[18 25 28 21 22];
```


2^a possibilità:
Mediante l'istruzione `struct`: il nome del campo è seguito dal contenuto che si vuole assegnare al campo

```
>> studente=struct('nome','Mario Rossi',...
                   'matricola',012345,...
                   'voti',[28 25 18 24 30]);
```

e per aggiungere un nuovo elemento alla struttura:

```
>> studente(2)=struct('nome','Mario Rossi',...
                      'matricola',012345,...
                      'voti',[28 25 18 24 30]);
```

```
>> studente
1x2 struct array with fields:
    nome
    matricola
    voti
```

Oss. *Non viene piú stampato il contenuto della struttura.*

```
>> studente(2)
ans =
        nome: 'Bruno Bianchi'
    matricola: 54321
        voti: [18 25 28 21 22]
```

Per visualizzare i dati del secondo elemento della struttura:

```
>> studente.voti
ans =
    28    25    18    24    30
ans =
    18    25    28    21    22
```

Per visualizzare il campo `voti`

```
>> studente(1).voti(4)
ans =
    24
```

Per visualizzare il 4° elemento del campo `voti` *del 1° studente.*

Per **aggiungere** un nuovo campo alla struttura:

```
>> studente(1).anno=3;
```

Oss. Ad ogni elemento è aggiunto il campo `anno` con contenuto vuoto.

```
>> studente(2)
ans =
        nome: 'Bruno Bianchi'
    matricola: 54321
        voti: [18 25 28 21 22]
        anno: []
```

Per **cancellare** un campo della struttura

```
>> studente=rmfield(studente,'anno')
```

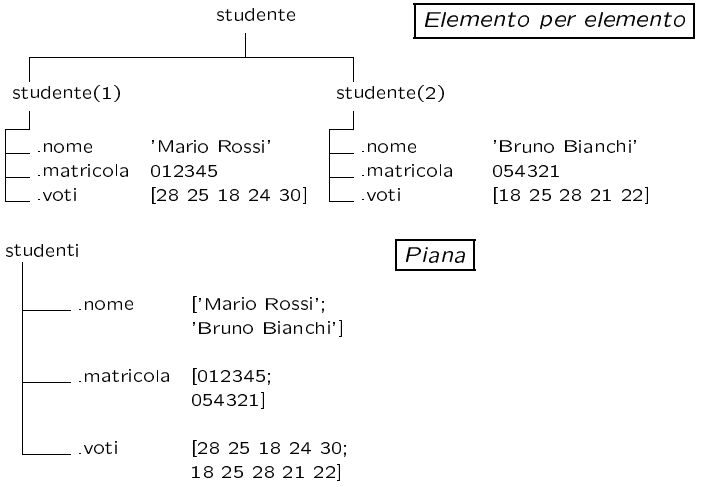

Su ogni singolo campo della struttura si possono svolgere operazioni.
Es. Calcolare la media dei voti di ogni singolo studente.

```
>> n=length(studente);
>> for i=1:n
media(i)=mean(studente(i).voti);
end
>> media
   25.0000   22.8000
```

Es. Calcolare la media dei voti tra tutti gli studenti.

```
media_glo=mean([studente.voti]);
>> media_glo
   23.900
```


Organizzazione dei dati nella struttura



Per costruire la struttura secondo l'organizzazione piana:

```
>> studenti.nome=strvcat('Mario Rossi','Bruno Bianchi');
>> studenti.matricola=[012345;054321];
>> studenti.voti=[28 25 18 24 30;18 25 28 21 22];
>> studenti
         nome: [2x13 char]
    matricola: [2x1 double]
         voti: [2x5 double]

>> studenti.nome
ans =
Mario Rossi
Bruno Bianchi
```

Il comando `strvcat` concatena verticalmente delle stringhe.

Per aggiungere nuovi elementi alla struttura (organizzazione piana):

```
>> studenti.nome=strvcat(studenti.nome,'Luigi Ferrari');
>> studenti.matricola=[studenti.matricola;034123];
>> studenti.voti=[studenti.voti;23 30 28 30 26];
```

• Organizzazione *elemento per elemento*: utile se si deve operare su tutte le informazioni di un singolo elemento.

• Organizzazione *piana*: utile se si deve operare contemporaneamente su tutti i dati di un campo.

La scelta di uno o dell'altro tipo di organizzazione dei dati dipende dal tipo di lavoro che si deve fare.

Classi ed oggetti

La programmazione ad oggetti può migliorare significativamente il riutilizzo dei codici e rendere più facile la manutenzione e l'aggiornamento di una libreria di programmi da parte di più programmatori.

I concetti di base della programmazione ad oggetti sono:

• INCAPSULAZIONE: l'oggetto deve essere quanto più possibile indipendente: tutto ciò di cui l'oggetto ha bisogno deve essere interno all'oggetto stesso (un oggetto può contenere dati e funzioni su di esso).

• EREDITARIETA': si possono creare nuovi oggetti che ereditano tutte le proprietà di altri oggetti e che, in più, hanno nuove funzionalità.

• OVERLOADING (o POLIMORFISMO): sugli oggetti possono essere definite operazioni che già esistono per dati di tipo diverso precedentemente definiti.

• AGGREGAZIONE: un determinato oggetto può essere parte di un altro oggetto.

Con le classi e gli oggetti si possono aggiungere nuovi tipi di dati e nuove operazioni all'ambiente MATLAB.

Una *classe* di variabili "dice" come sono fatte quelle variabili e quali tipi di operazioni si possono fare su di esse.

Un *oggetto* è una variabile di una classe specifica.

Un *metodo* è una particolare operazione ("function") che agisce sugli oggetti di una classe.

Gli oggetti sono memorizzati in *strutture*. I campi di queste strutture sono visibili solo attraverso i metodi della classe.

Quando si definisce una classe di oggetti, bisogna anzitutto definire alcuni metodi fondamentali:

- *constructor*: per creare gli oggetti della classe
- *display*: per visualizzare gli oggetti della classe
- *set* e *get*: per accedere alle proprietà degli i oggetti
- *subsref* e *subsagn*: per leggere o definire parte di un oggetto tramite degli indicii (es: `A(i)=5`).
- *converters*: per convertire un oggetto in un determinato tipo di dato di MATLAB.

Esempio: la classe dei polinomi.
Costruire la classe *@polinomio*, defininendo alcuni metodi sugli oggetti polinomi: somma algebrica, prodotto, derivazione, integrazione, calcolo delle radici, disegno.

0. In `C:\temp` creare una cartella di nome `Classi` ed al suo interno un'altra cartella di nome `@polinomio`, dove salvare tutti i *metodi* di questa classe.

1. Scrivere il *constructor* della classe: è una function con lo stesso nome della classe (`polinomio.m`). Serve per specificare come sono fatti gli oggetti di questa classe.

La function *constructor* deve contemplare in input:
1. nessun argomento (crea un oggetto vuoto)
2. argomento/i di tipo oggetto (copia l'oggetto dato in input)
3. argomento/i di tipo dati (costruisce l'oggetto)

Vogliamo costruire un polinomio a partire dai coefficienti:

`a=[2 3 0 1]` $\longrightarrow p(x) = 2x^3 + 3x^2 + 1$

```
a=[2 3 0 1]      ---->       p.c       ----->    p
   dato                   struttura            oggetto
```

```
function p=polinomio(a)
% POLINOMIO     costruttore della classe polinomio
% p=polinomio(a)   a=vettore dei coefficienti
%                  in ordine di potenze di x decrescenti
if nargin == 0
p.c=[0]; p=class(p,'polinomio');
elseif isa(a,'polinomio')
p=a;
else
p.c=a(:).'; p=class(p,'polinomio');
end
```

Per costruire l'oggetto polinomio:

```
>> addpath C:\temp\Classi

>> p=polinomio([2 3 0 1])
p =
        polinomio object: 1-by-1

>> p1=polinomio([2 1])
p1 =
        polinomio object: 1-by-1

>> whos
  Name            Size         Bytes  Class

  p1              1x1            168  polinomio object
  p               1x1            168  polinomio object
```


2. Scrivere un metodo per scrivere a video l'oggetto nel formato desiderato: `display.m`.
Questa function viene invocata automaticamente da MATLAB ogni volta che si chiede di stampare a video il contenuto dell'oggetto creato:

```
>> p  ⏎
```

```
function display(p)
% POLINOMIO/DISPLAY
% Stampa a video del polinomio
disp(' ');
disp([inputname(1),'(x) = ',char(p)])
disp(' ');
```

`char` è una function che serve per convertire l'oggetto polinomio in una stringa.


```
function s=char(p)
% POLINOMIO/CHAR
% CHAR(p) e' la stringa rappresentante p.c
if all (p.c == 0)
 s='0';
else
 n=length(p.c)-1; s=[];
 for a=p.c;
  if a~=0
   if ~isempty(s)
    if a>0; s=[s ' + ']; else; s=[s ' - ']; a=-a; end
   end
   if a~=1 | n==0
    s=[s num2str(a)]; if n>0; s=[s '*']; end
   end
   if n>=2; s=[s 'x^' int2str(n)]; elseif n==1; s=[s 'x']; end
  end
  n=n-1; end; end
```


Copiare `char.m` in
`C:\temp\Classi\@polinomio\char.m`

```
>> p

p(x) = 2*x^3 + 3*x^2 + 1

>> p1

p1(x) = 2*x + 1
```

Function `subsref.m`

È la function invocata da MATLAB quando viene dato il comando `a(i)` (`a`=array, `i`=indice).
Si può dare a questa function il significato che più è opportuno per il tipo di oggetto. Nel caso dei polinomi: `p(3)` sigifica "valutare `p(x)` in `x=3`".

```
function b=subsref(p,s)
```

p è l'oggetto che si vuole referenziare

s è una struttura con due campi: `s.type` e `s.subs`

se `s.type` $=()$ $\Rightarrow$ s è un dato numerico
$=\{\}$ $\Rightarrow$ s è una cella
$=.$ $\Rightarrow$ s è una struttura

`s.subs` è una cella o una stringa contenente i valori che servono per referenziare l'oggetto.


```
function b=subsref(p,s)
% POLINOMIO/SUBSREF
switch s.type
case '()'
% s e' un dato numerico
  ind=s.subs{1};
  b=eval(strrep(char(p),'x',num2str(ind)));
otherwise
  error('Referenza errata')
end
```

Si ha:

```
>> p(3)
ans =
    82
```


3. **Overloading** delle operazioni.

Ridefinire alcune operazioni (es. +, -, *) per gli oggetti polinomi, in modo da poter dare i comandi:

```
p+p1
p-p1
p*p1
diff(p)
double(p)
plot(p)
.....
```


Per costruire la somma si deve scrivere la function `plus.m`:

```
function r=plus(p,q);
% POLINOMIO/PLUS Somma di due polinomi
k=length(q.c)-length(p.c);
p1=[zeros(1,k) p.c];
q1=[zeros(1,-k) q.c];
r=polinomio(p1+q1);
```

Per fare la somma:

```
>> q=p+p1

q(x) = 2*x^3 + 3*x^2 + 2*x + 2
```

Analogamente per la differenza, bisogna scrivere la function `minus.m`. (modificare opportunamente `plus.m`).

Per il prodotto tra due polinomi: `mtimes.m`

```
function r=mtimes(p,q);
% POLINOMIO/MTIMES Prodotto di due polinomi

r=polinomio(conv(p.c,q.c));
```

`conv` è una function MATLAB che genera il vettore dei coefficienti del polinomio prodotto a partire dai coefficienti dei polinomi operandi.

```
>> q=p*p1

q(x) = 4*x^4 + 8*x^3 + 3*x^2 + 2*x + 1
```


Altri metodi

Function `double.m`: per estrarre dall'oggetto polinomio il vettore dei coefficienti.

```
function c = double(p)
% POLINOMIO/DOUBLE  c = double(p)
c=p.c;
```

Function `diff.m`: per calcolare il polinomio derivata di un dato polinomio.

```
function q=diff(p)
% POLINOMIO/DIFF q = diff(p),  derivata di p(x)
c=p.c;
n=length(c)-1;
q=polinomio(c(1:n).*(n:-1:1));
```


Function `primitiva.m`: per calcolare il polinomio primitiva di un dato polinomio.

```
function q=primitiva(p)
% POLINOMIO/PRIMITIVA q = primitiva(p)
c=double(p.c);
n=length(c)-1;
num=[c(1:n+1), 0]; den=[(n+1:-1:1),1];
q=polinomio([num./den]);
```

$$\text{Se } p(x) = c_1 x^n + c_2 x^{n-1} + ... + c_{n+1}$$

$$\Rightarrow q(x) = \frac{c_1}{n+1} x^{n+1} + \frac{c_2}{n} x^n + ... + c_{n+1} x$$

La classe `@polinterv`

Vogliamo costruire una nuova classe che **erediti** alcuni metodi dalla classe `@polinomio` e che ne abbia altri propri.

Vogliamo che l'oggetto `polinterv` sia un polinomio corredato dei valori estremi di un intervallo reale su cui si vuole calcolare l'integrale del polinomio stesso.

Es: $\int_{x_1}^{x_2} p(x)dx$

L'oggetto è individuato da $p(x)$, x_1 e x_2.

Creiamo gli oggetti `polinterv` come **figli** degli oggetti `polinomio` $\Rightarrow$ **ereditarietà semplice**.

Oss. Si parla di **ereditarietà multipla** quando un oggetto è figlio di oggetti di classi diverse (eredita i metodi di tutte le classi di cui è figlio).

0. Creare il direttorio `C:\temp\Classi\@polinterv`

1. Creare la function `polinterv.m` che definisca la classe.

Oss. L'istruzione che definisce che un oggetto `p` di tipo `polinterv` è figlio di un oggetto `a` di tipo `polinomio` è:

```
a=polinomio([...]);
p=class(p,'polinterv',a);
```

oggetto=class(struttura,'nome classe',padre)


```
function p=polinterv(varargin)
% POLINTERV, p=polinterv(vett_coeff,xmin,xmax)
switch nargin
case 0 % nessun input
  p.xmin=0; p.xmax=0; a=polinomio;
  p=class(p, 'polinterv',a);
case 1 % un input che e' gia' un oggetto polinterv
  if(isa(varargin{1},'polinterv'))
    p=varargin{1};
  else
    error('L'input non e' un oggetto polinterv')
  end
case 3 % tre input: creo l'oggetto
  p.xmin=varargin{2}; p.xmax=varargin{3};
  a=polinomio(varargin{1}); p=class(p,'polinterv',a);
otherwise
  error('Numero sbagliato di input')
end
```


Oss. Un oggetto `p` di tipo `polinterv` ha 3 campi:

```
p.xmin
p.xmax
p.polinomio   campo contenente l'oggetto padre
```


Per definire un oggetto polinterv:

```
>> addpath C:\temp\Classi
>> pint=polinterv([4 2 0 1],0,1)

pint = 4*x^3 + 2*x^2 + 1
```

oppure

```
>> p=polinomio([4 2 0 1]);
>> pint=polinterv(p,0,1);
```

Oss. `pint` è anche un polinomio e viene utilizzato il metodo `polinomio\display` per stamparlo. Si può scrivere un nuovo metodo `polinterv\display` che stampi anche gli altri due parametri.

```
function display(p)
% POLINTERV/DISPLAY, stampa pol. con estremi intervallo
disp(' ');
disp([inputname(1),'(x) = ' char(p) ' su ['...
 num2str(p.xmin) ',' num2str(p.xmax) ']' ])
disp(' ');
```

```
>> addpath C:\temp\Classi
>> pint
pint(x) = 4*x^3 + 2*x^2 + 1 su [0,1]

>> pint2=polinterv([2 1],-1,2);
```

Posso fare somma, sottrazione, prodotto tra questi oggetti, sfruttando i metodi della classe `@polinomio`. I risultati di queste operazioni sono oggetti della classe `@polinomio`.

```
>> qint=pint+pint2
qint(x) = 4*x^3 + 2*x^2 + 2*x + 2
```


Costruiamo la function che calcola l'integrale $\int_{x_1}^{x_2} p(x)dx$

```
function I=integro(p)
% POLINTERV/INTEGRO, I = integro(p)
%  calcola l'integrale del polinomio p(x)
% su [p.xmin,p.xmax]
a=p.xmin; b=p.xmax;
q=primitiva(p.polinomio);
I=q(b)-q(a);
```

Quindi:

```
>> P=integro(pint)
P =
    2.6667
```


La classe `@razionale`

Vogliamo costruire una nuova classe che **aggreghi** degli oggetti della classe `@polinomio`.

Definiamo un oggetto `razionale` come una funzione fratta in cui numeratore e denominatore sono polinomi.

$$r(x) = \frac{x^3 - 1}{x^2 + 1}$$

0. Creare il direttorio `C:\temp\Classi\@razionale`

1. Creare la function `razionale.m` che definisca la classe.


```
function q=razionale(a1,a2)
% RAZIONALE, costruttore classe razionale q(x)=p1(x)/p2(x)
if nargin == 0
q.num=[]; q.den=[]; q=class(q,'razionale');
elseif isa(a1,'razionale')
q=a1;
else
q.num=polinomio(a1); q.den=polinomio(a2);
q=class(q,'razionale');
end
```

Per costruire un oggetto razionale:

```
>> p=polinomio([1 0 0 -1]); q=polinomio([1 0 1]);
>> r=razionale(p,q)
r =
        razionale object: 1-by-1
```

oppure

```
>> r=razionale([1 0 0 -1],[1 0 1])
```

2. Function display

```
function display(r)
% RAZIONALE/DISPLAY, stampa a video della funzione razionale
disp(' ');
disp([inputname(1),'(x) = (' char(r.num) ')/(' char(r.den) ')'])
disp(' ');
```

```
>> r=razionale(p,q)
r(x) = (x^3 - 1)/(x^2 + 1)
```


La function subsref

```
function b=subsref(p,s)
% RAZIONALE/SUBSREF
switch s.type
% s e' un numero
case '()'
  ind=s.subs{1};
  b=eval(strrep(char(p.num),'x',num2str(ind)))/...
  eval(strrep(char(p.den),'x',num2str(ind)));
% s e' un campo (il campo dei coeff. del num. o den.)
case '.'
  switch s.subs
    case 'num'; b=(p.num);
    case 'den'; b=(p.den);
  end
otherwise
  error('Messaggio d'errore....')
end
```



```
>> r(2)
ans =
    1.4000

>> r.num
ans(x) = x^3 - 1

>> r.den
ans(x) = x^2 + 1
```


Per vedere l'elenco dei metodi definiti per una classe:

```
>> methods polinomio
Methods for class polinomio:

char      display   minus     polinomio primitiva
diff      double    mtimes    plus      subsref
```

Alcuni metodi standard (oltre al constructor):

```
display, subsref, subasgn,
subsindex, double, char
```

Per ognuno di essi:

```
>> help nome_metodo
```

(sono metodi standard di MATLAB).

Nomi di function associati agli operatori fondamentali di MATLAB:

a+b	plus(a,b)
a-b	minus(a,b)
a*b	mtimes(a,b)
a.*b	times(a,b)
a<b	lt(a,b)
a&b	and(a,b)
a.'	transpose(a)
a:b	colon(a,b)

Per un elenco piú dettagliato:

```
>> help *
```


- Per cancellare un oggetto: `>> clear` *oggetto*

- L'ereditarietà è stabilita nella classe figlio, creando l'oggetto padre e quindi richiamando la function `class`

- L'oggetto figlio contiene un oggetto padre in un campo che ha il nome dell'oggetto padre

- L'aggiornamento di un oggetto deve essere fatto tramite i metodi `set` e `get`

- in MATLAB non c'è l'equivalente di una classe astratta

- in MATLAB non c'è l'equivalente di un'interfaccia Java

...

Cell arrays

Un *cell array* è un array i cui elementi sono *cells*, ovvero contenitori di array di tipo numerico, stringa o altro.

cell 1,1 $\begin{bmatrix} 3 & 4 & 2 \\ 9 & 7 & 6 \\ 8 & 5 & 1 \end{bmatrix}$	**cell 1,2** $\begin{bmatrix} \text{'Mario Rossi'} \\ \text{'Bruno Bianchi'} \end{bmatrix}$
cell 2,1 $\begin{bmatrix} 2+3i & 8-i \\ 4+.2i & -3i \end{bmatrix}$	**cell 2,2** $\begin{bmatrix} 23 & 32 & 21 \end{bmatrix}$

Per creare un cell array:

```
>> A(1,1)={[3  4 2; 9  7  6; 8  5  1]};
>> A(1,2)={strvcat('Mario Rossi','Bruno Bianchi')};
>> A(2,1)={[2+3i  8-i; 4+.2i  -3i]};
>> A(2,2)={[23 32 21]};
>> A
A =
    [3x3 double]    [2x13 char  ]
    [2x2 double]    [1x3  double]
```

oppure:

```
>> A{1,1}=[3  4 2; 9  7  6; 8  5  1];
>> A{1,2}=strvcat('Mario Rossi','Bruno Bianchi');
>> A{2,1}=[2+3i  8-i; 4+.2i  -3i];
>> A{2,2}=[23 32 21];
```

Per leggere la cella $A(1,1)$:

```
>> A{1,1}
ans =
     3     4     2
     9     7     6
     8     5     1
```

Per leggere l'elemento $(2,2)$ della cella $A(2,1)$:

```
>> A{2,1}(2,2)
ans =
        0 - 3.0000i
```

Per leggere tutto il cell array:

```
>> A{:}
```