Manuale Java

di Michele Sciabarrà

Sintassi

In questo capitolo faremo una panoramica della sintassi.
Vedremo anche un importantissimo aspetto: la distinzione tra tipi primitivi e oggetti.

- Java è un linguaggio imperativo orientato agli oggetti.
- Quindi contiene espressioni che vengono gestite tramite comandi.
- I comandi vengono incapsulati in metodi
- I metodi vengono contenuti in classi
- A loro volta le classi vengono raccolte in package.

Sintassi Commenti

Per cominciare vediamo come si inseriscono commenti nei programmi Java:

```
/* commento */
```

- Tutti i caratteri compresi tra /* e*/ sono ignorati.

```
// comment
```

- Tutti i caratteri successivi a // fino a fine linea sono ignorati.

```
/** comment */
```

- Come /* */, eccetto che il commento viene usato con il tool `javadoc` per creare automaticamente la documentazione dai sorgenti .

Sintassi espressioni

Un programma in Java è innanzitutto composto da *espressioni*:

```
a+1
```

Una espressione si può suddividere in:

- *Costanti*: `1` è una costante
- *Operatori*: `+` è un operatore
- *Variabili*: a è una variabile

Sintassi comandi

I *comandi* contengono espressioni.

I comandi possono essere:

- Semplici (una espressione seguita da un ;):

```
a+1;
```

- Composti (un comando contenente un altro comando):

```
if (a>1)
   b=b+1;
```

Sintassi dichiarazioni

Le *dichiarazioni* danno il tipo ad una variabile.

```
int i;
```

Una dichiarazione può essere seguita da una espressione di inizializzazione (opzionale):

```
String s = "hello";
```

Sintassi metodi

Un *metodo*, quando si dichiara è simile ad una funzione matematica: ha dei parametri e ritorna un valore.

```
int sum(int x, int y)
```

Notare che *f* è un metodo che prende due interi come parametri e ritorna un intero.

Un metodo è seguito da un comando che definisce il suo comportamento:

```
int sum (int x, int y)  {
  return x+y;
}
```

Una particolarità dei metodi è che si trovano *sempre* dentro una classe.

Infatti i metodi hanno sempre un *contesto* (o ambiente):

```
int sumk (int x) {
    return x+k;
}
```

k appartiene al contesto

In un metodo possono esserci dichiarazioni di variabili (locali al metodo):

```
int sumk (int x, int y) {
    int r=x+y;
    return r;
}
```

Sintassi classi

Le classi contengono campi e metodi:

```
class Sum {
    int k=1;             // campo
    int sum3(int x) {    // metodo
        int  h = 2;
        return x+h+k;
    }
}
```

Un campo (*k)* è una cosa **diversa** da una variabile locale (*h)* e da parametro di un metodo (*x)*.

Tipi di dato

Come abbiamo detto, ci sono due "gruppi" di tipi di dato:

- Primitivi
- Oggetti

Primitivi:

- Sono *fissi*
- Una "istanza" di tipo primitivo è generata da una *costante*
- Si agisce su di essi con gli *operatori*
- Una variabile "contiene" un tipo primitivo (il concetto tradizionale di variabile)

Oggetti:

- Sono in numero illimitato (creati dal programmatore)
- Una "istanza" di oggetto è generata da un *costruttore*
- Si agisce su di essi con i *metodi*
- Una variabile "riferisce" un oggetto: è un puntatore!

Confronto Primitivi e Oggetti

<table>
<tr>
<td>

```
int a = 1;
```

Utilizzata una costante (1)

</td>
<td>

```
Stack s=new Stack();
Stack t= new Stack();
```

Utilizzato un *costruttore*

</td>
</tr>
<tr>
<td>

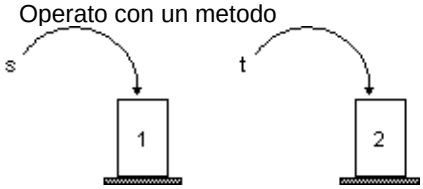
```
int b=a+1;
```

Operato con un operatore (+)

</td>
<td>

```
s.push(1); t.push(2);
```

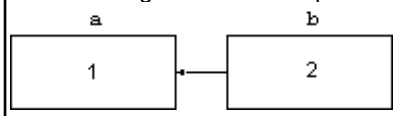
Operato con un metodo



</td>
</tr>
<tr>
<td>

```
a = b;
```

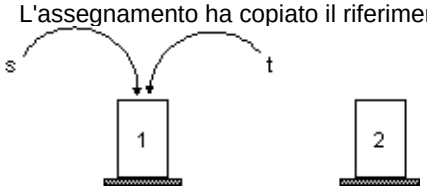
L'assegnamento ha copiato il valore



</td>
<td>

```
t = s;
```

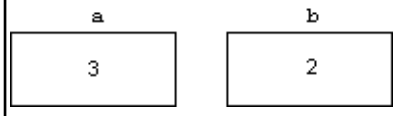
L'assegnamento ha copiato il riferimento



</td>
</tr>
<tr>
<td>

```
a = 3;
```

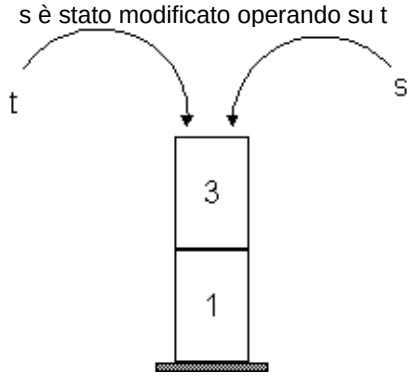
b *non* è stato modificato



</td>
<td>

```
t.push(3);
```

s è stato modificato operando su t



</td>
</tr>
</table>

Tipi Primitivi

Dimensione fissa per tutte le Java Virtual Machine:

Tipi interi:
`byte 8 bit`
`short 16 bit`
`int 32 bit`
`long 64 bit`
	Tipo booleano:
	`boolean 1 bit`
	Tipo carattere:
	`char 16 bit` (carattere Unicode)
	Tipi floating−point:
	`float 32 bit IEEE`
	`double 64 bit IEEE`

Espressioni

Per calcolare le espressioni occorre considerare:

- Precedenza, ovvero chi prevale tra due operatori diversi:

```
a + 2 * 3   // come?
a + (2 * 3) // * ha la precedenza su +
```

- Associatività ovvero chi prevale con un solo operatore:

```
a + b + c      // come?
(a + b ) + c   // + associa da destra
a = b = c      // come?
a = ( b = c)   // = associa da sinistra
```

- Effetti collaterali:

```
a = 1 // espressione, vale 1
      // come effetto collaterale di =, a viene modificata
a = b = 1
      // ad a assegno il valore di (b = 1)
a = int b = 1
      // ERRORE! int b (inizializzazione) non è una espressione
```

Costanti booleane

Le costanti booleane *non* sono interi (come in altri linguaggi).
Sono le due costanti:

- `true`
- `false`

Costanti carattere

Le costanti carattere sono caratteri tra tra virgolette singole:

```
'a' '1' '\n'
```

Alcuni caratteri sono esprimibili con una sequenza di due caratteri che comincia con \ (backslash):

```
Backslash                    \\
Continuazione                \
```

```
Spazio indietro                \b
Ritorno del carrello           \r
Salto Pagina                   \f
Tabulazione orizzonatale       \t
Nuova linea                    \n
Virgoletta semplice            \'
Doppia virgoletta              \*
Carattere ottale               \ddd
Carattere unicode              \udddd
```

- Notare la sequenza `\ddd` che serve ad esprimere un carattere ASCII scrivendone il suo codice *in ottale*:

  dopo \ ci vogliono **da 1 a 3 cifre ottali**
- Notare la sequenza `\udddd` che serve ad esprimere un carattere UNICODE scrivendone il suo codice *in esadecimale*:

dopo \u ci vogliono **esattamente 4 cifre esadecimali**

Costanti Numeriche

Intere:

- decimali: `17 −1`
- esadecimali: `0xff` cifre 0−9,A−F, `0x` o `0x` in testa
- ottali: `0177` cifre 0−8, `0` in testa

Intere lunghe:

- decimali `17l, −1L`

Non ci sono costanti short o byte

- o se ne accorge il compilatore:

  ```
  byte b = 17;
  ```
- oppure viene richiesta una conversione:

```
b = (byte)257;
```

Floating point:

- `19.4 19f` float, in formato decimale o con la f/F in coda
- `17d 17D` double con la d/D in coda

Operatori

Gli operatori agiscono sui *tipi primitivi:*

Relazionali:

```
> >= < <= |= ==
```

Aritmetici:

```
+ − * / %
```

Binari:

```
& | ^ >> << <<<
```

Logici:

```
&& || ?:
```

Incremento:

```
++ −−
```

Assegnamento:

```
= += −= *= /= ...
```

Regola Promozione

Nei calcoli:

- ogni valore intero inferiore ad int diventa int
- ogni valore float diventa double
- un operatore opera solo su valori della stessa dimensione
- tra due operandi, (a + b) il "più grosso" comanda e l'altro viene ingrandito
- per assegnare a valori inferiori bisogna "convertire" (troncando)

Esempio:

```
byte b=1;
short s=2;
long l=3;
double d=4.0;
int t=b+s;          è int (32 bit)
t+l                 è long (64 bit)
l+d                 è double (64 bit)
byte n=t+l;         ERRORE
byte n=(byte)t+l;   OK
```

Operatori in realtà

- Sono “funzioni overloaded”:

  `*` prende due `int` e ritorna un `int`
  ma anche `*` prende due `double` e ritorna un `double`
- Analoghi (ma immutabili) ai metodi overload

`int mult (int x, int y)` prende due int e ritona un int
`double mult (double x, double y)` stesso nome, ma prende due double e ritorna un double

Operatori aritmetici

- `+ - * / %` (binari)

Somma, sottrazione, prodotto, divisione e modulo.
Prendono due argomenti, di tipo int o float e ritornano un int o float

- `- +` (unari)

Il meno unario inverte il segno di un int o di un doble
Il più unario non fa nulla ma serve per poter scrivere +4 senza errore di sintassi.

Aritmetica Floating Point

Java conosce l’analisi matematica!

- Segue lo standard IEEE 754
- Esistono gli infiniti (`Float.POSITIVE_INFINITY Float.NEGATIVE_INFINITY` , etc)
- Esiste la *forma indeterminata* `Float.NaN` (Not A Number)

x	**y**	**x/y**	**x%y**
finito	+/− 0	+/− infinito	NaN

NaN	+/− infinito	NaN	NaN
+/− 0	finito	+/− infinito	NaN
+/− infinito	+/− infinito	NaN	NaN

Operatori Relazionali

`> >= < <= == !=`

- Maggiore, MaggioreUguale, Minore, MinoreUguale, Uguale e Diverso
- Notare che Uguale è *due* segni '=' (mentre l'assegnamento è *un* segno '=')
- Operano su interi e double
- Uguale e Diverso operano anche su booleani

Attenzione al comportamento con l'aritmetica floating point di Java

- −infinito < finito < + infinito
- + infinito == +infinito
- NaN != NaN

Poichè una forma indeterminata non è comparabile con una forma indeterminata,
per vedere se *a* è *NaN* non posso scrivere *a==NaN*
ma devo scrivere *!a==a* (l'unico caso in cui *a* sia diversa da se stessa è che sia una *NaN*)

Operatori Bit−a−bit

& (and) | (or) ^ (xor)

- operano su interi
- and or e exclusive−or bit−a−bit sui bit dei valori interit
- operano anche su booleani, operando su un solo bit
- Notare che se faccio `f() & g()` viene chiamato *f()* e viene chiamato *g()* **sempre**

~ (not)

- opera solo su interi
- inverte ogni bit

`! (not)`

- opera su booleani
- inverte il valore true/false

`<< >> >>>`

- operano su interi
- >> shift con estensione di segno: `−2>>1 == −1`
  - ♦ utile quando si considera l'intero un valore con segno
- >>> shift senza estensione di segno: `−2>>>1 == 2147483647`
  - ♦ utile quando si considera l'intero una maschera di bit

Operatori Logici

&& (and)

- opera su valori booleani
- comportamento short − circuit:

valore determinato se il primo operando è false
in tal caso non viene valutato il secondo E' utile:

```
c != −1 && c = in.read()
```

se c'è EOF (−1), non viene letto un altro carattere

|| (or)

- comportamento short − circuit:

valore determinato se il primo operando è `true`
n tal caso non viene valutato il secondo E' utile:
`n ? table.min() || n ? table.max()`
si risparmia un calcolo (pesante) del max

? : operatore ternario (l'unico)

- il primo valore deve essere un valore booleano, gli altri due possono essere quasiasi tipo ma devono essere lo stesso tipo

comportamento short − circuit: max: (a > b) ? f() : g() (se a>b chiamo f() altrimenti chiamo g()) E' utile:
`float sqrtmax(float a, float b) { return a ? b ? Math.sqrt (a):Math.sqrt(b); }`
calcola la radice una volta sola

Operatori Incremento

- Classici del C : `++` incremento di 1 `−−` decremento di 1
- operano su interi e float
- Prefissi o postfissi:

`a = 1 ;`
`b = a++;`
ora `b==1 e a==2`
	`a = 1 ;`
	`b = ++a ;`
	ora `b==2 e a==2`

Operatori di Assegnamento

`a = b` è una espressione con effetti collaterali:

- modifica a
- ritorna b
- `a = b = c` equivale a `a = (b = c)`
- b=c modifica b e ritorna il valore di c che serve a modificare a. Il tutto vale c, valore scartato

`a += b`
- equivale a `a = a + b`
- però `a[f()] += b` non equivale a `a[f()]=a[f()]+b`
- l'intero valore del primo operando viene calcolato una volta sola (e quindi f() chiamato una volta sola)

altri operandi, stesso comportamento (cambia l'operatore di calcolo)
`−= *= /=`
`&= |= ^=`
`<<= >>= >>>=`

Comandi

I comandi di Java possono essere distinti nelle seguenti categorie:

- semplici e blocchi
- condizionali: `if − else, switch`
- di ciclo: `while, do − while, for`
- di interruzione di ciclo: `break, continue`
- ritorno di valori: `return`
- gestione eccezioni: `try − catch − finally`

Semplici e Blocchi

- se si ha una espressione

```
a = 1
```

aggiungendo un ';' diventa un comando (semplice)

```
a = 1 ;
```

- se si hanno due comandi

```
a=1; b=2;
```

acchiudendoli tra graffe diventano un solo comando (blocco)

```
{ a=1; b=2; }
```

- Notare che il punto e virgola ci vuole sempre, anche nell'ultima istruzione di un blocco

Condizionale if

```
if(<condizione>)
 <comando>
else
 <comando>
```

- la `<condizione>` deve essere una espressione booleana
  - non può essere intera (come in C)

quindi non `if(a)` ma `if(a!=0)`

- l'`else` lega l'`if` più vicino altrimenti si devono usare le graffe

```
if(<condizione>) //1
 if(<condizione>) //2
    <comando>
 else // riferito a 2
  <comando>
```

```
if(<condizione>) //1
{
    if(<condizione>) //2
        <comando>
} else  // riferito a 1
  <comando>
```

Condizionale switch

```
switch (<condizione>) {
 case <val1>:
 case <val2>:
    <comando>
 break;
 case <val>:
  …
 break;
 default:
 …
 break;
}
```

- Viene valutata la `<condizione>` che deve dare un risultato **intero**
- Si salta al primo `vali` corrispondente al valore nei `case`

- **Attenzione:** una volta eseguito un salto, il flusso *prosegue* all'interno dello switch.

  Non si esce dallo switch al case successivo. Questo è utile per impilare più case.
- Se si incontra un *break* si va alla fine dello switch
- Se nessuno dei case è soddisfatto si salta al default:

Esempio:

```
n=0;
switch (c) {
 case 1:
 case 2:
    n += 1;
 case 3:
      n += 2;
 break;
 default:
      n += 4;
 break;
}
```

- Se all'inizio *c* vale 1 o 2, *n* alla fine vale 3
- Se all'inizio *c* vale 3 , *n* alla fine vale 2
- Se all'inizio *c* vale 4, *n* alla fine vale 4

Ciclo while

```
while(<condizione>)
  <comando>
```

- La condizione deve essere una espressione boolena
- Il comando viene eseguito finché è vera la condizione
- La condizione viene valutata per prima, per cui il comando può anche essere eseguito 0 volte

Ciclo do−while

```
do
 <comando>
while(<condizione>) ;
```

- La condizione deve essere una espressione boolena
- Il comando viene eseguito finché è vera la condizione
- Il comando viene eseguito per primo, per cui il comando verrà sempre eseguito almeno una volta

Ciclo for

```
for(<inizio>;
    <condizione>;
    <incremento>)
                <comando>
```

- Simile al `while`
- Per prima cosa viene eseguita l'espressione di <*inizio*>
- Poi viene valutata la <*condizione*>
- Viene eseguito il <*comando*>
- Viene eseguita l'espressione di <*incremento*>

Analogo a:

```
<inizio>;
while(<condizione>) {
  <comando>;
  <incremento>;
}
```

Interruzione di ciclo

`break`

- Consente di interrompere cicli e switch

`continue`

- Consente di proseguire cicli dal punto in cui si è

```
while(...) {
  if(...)
    break;
  if(...)
   continue;
}
```

In pratica il `break` equivale ad un salto alla fine del ciclo, mentre il `continue` equivale ad un salto alla fine del ciclo.

Label

- All'inizio dei cicli si possono apporre label
- `break` e `continue` possono avere come parametro una label
- In questo modo è possibile uscire o continuare da cicli annidati

Esempio: ricerca in più file

```
loop:
 while(<ci-sono-file>) {
    while (<ci-sono-righe>) {
        if(<non-c'è-in-questo-file>)
           continue loop;
        if(<trovato>)
           break loop;
     }
}
```

Eccezioni

- Le chiamate di metodi possono generare "eccezioni"
- Le eccezioni si propagano causando la terminazione del programma con un messaggio di errore se non vengono gestite
- Saranno trattate in maggior dettaglio con le classi.
- Per il momento vediamo come ignorarle (stampando informazioni su dove è avvenuta l'eccezione.

```
try {
 <comando-che-genera-eccezione>
} catch (Exception ex) { ex.printStackTrace(); }
```

- In generale è opportuno propagare tutte quelle che non si può gestire (anche se non sempre è possibile). Il programma si interromperà più facilmente in fase di messa a punto ma risulterà più resistente alle condizioni di errori più avanti.
- Quando si dichiara un metodo che può sollevare eccezioni (generalmente chiamando altri metodi), si può "lasciar passare" le eccezioni aggiungendo `throws Exception` nella dichiarazione del metodo.

```
class C {
    void method(int x) throws Exception {
              <comando-che-genera-eccezione>
    }
}
```

Array e Stringhe

- Le Stringhe e gli Array sono a tutti gli effetti degli oggetti. Gli oggetti si creano con i costruttori e si opera con i metodi
- Sono però molto usati, e hanno una sintassi speciale per la loro costruzione, e alcuni metodi che vengono chiamati con una sintassi da *operatori*.
- Si tratta di *zucchero sintattico* che permette di operare più agevolemente su di essi.

Array

- Gli array sono degli oggetti.
- Gli array possono essere di tipi primitivi o di oggetti.
- Vengono dichiarati mettendo [] dopo il nome dell'oggetto, o mettendo [] dopo il nome della variable, o tutti e due.
- Essendo gli array oggetti, ci possono essere array di array.

```
int[] ax;      // array di interi
String ax[];   // array di stringhe
Object[] ay[]; // array di Object
int[][] aay;   // array di array di interi
```

Array

- Gli array hanno un campo `length` che ne dice la lunghezza. `ax.length` dà il numero di elementi presenti nell'array.
- Si accede agli elementi di un array utilizzando l'operatore `[]`: con `ax[i]` si accede all' i-simo elemento di `ax`. Notare che gli indici vanno da 0 a `ax.length-1`

```
class C {
    void main(String[]args) {
          for(int  i=0; i<args.length; i++)
               System.out.println(args[i]);
     }
}
```

Creare Array

- Dichiarare un array significa dichiarare un *riferimento* ad **un oggetto che non è stato ancora creato**. Per poter utilizzare l'oggetto occorre creare l'array. Se l'oggetto non è stato creato il riferimento è nullo

```
int[] ax;
ax[2]; // solleva Null Pointer Exception
```

- Un array può essere costruito con *new* o mettendo tra graffe i valori di inizializzazione.

```
int[] ai = new int[3]; // un array di tre interi
int[] ai = { 0, 1, 2 } // un array di tre interi
//a[0]==0, a[1]==1, a[2]==2
```

- Un array di oggetti o un array di array crea degli array i cui elementi sono riferimenti a oggetti *nulli*, a meno che non siano stati creati ponendoli tra graffe.

```
Stack[] as = new Stack[3];
Stack t = s[2];  //ok
as[2].push()     // solleva Null Pointer Exception
Stack[] bs = { new Stack(), new Stack(), new Stack() };
bs[2].push()    // ok
```

Stringhe

- sono degli oggetti con “zucchero sintattico” da tipi primitivi:
- costanti stringa

```
String s = “hello”;
```

breve per

```
char[] tmp = { ’h’, ’e’, ’l’, ’l’, ’o’ };
String s = new String(tmp) ;
```

- operatore di concatenazione + :

```
s = s + "world";
```

breve per

```
s=s.concat("world");
```

- operatore di assegnamento +=:

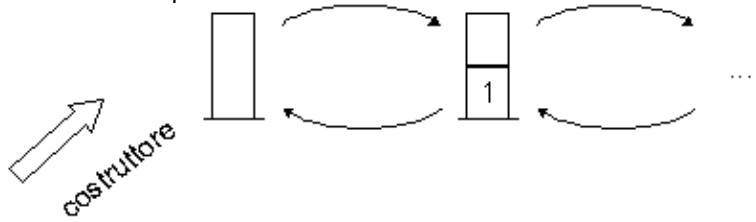
```
s += "world";
```

breve per

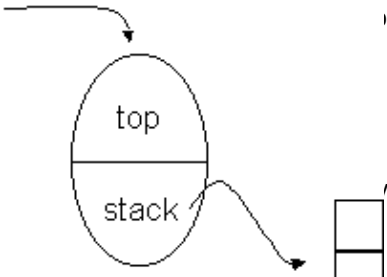
```
s = s+ "world";
```

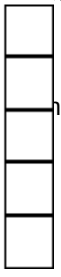
Classi

- Le classi implementano il principio dell'incapsulazione
- i campi reppresentano lo stato interno di un oggetto
- i costruttori inizializzano l'oggetto in uno stato noto
- i metodi permettono di cambiare lo stato in modo controllato



Classi come record





nsiderata come un semplice record.

o prima crearlo

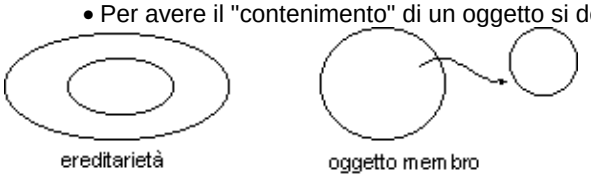
izializzarlo esplicitamente:

- Adesso possiamo utilizzarlo.

```
// operazione di push(1)
s.stack [s.top++]=1;
```

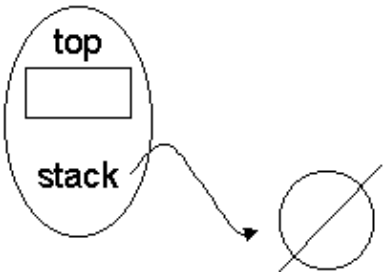
Classi e oggetti

- Gli elementi di una classe si chiamano campi
- I campi sono sempre:
  - tipi primitivi
  - riferimenti ad altri oggetti
- Per avere il "contenimento" di un oggetto si deve ricorrere all'ereditarietà.



- Allocazione di un oggetto:
  - crea i campi tipi primitivi
  - crea i riferimenti nulli ai sotto oggetti

```
Stack s= new Stack () ;
```



Costruttori

- Si può legare il codice di inizializzazione all'oggetto stesso:
- Ecco come evitare le inizializzazioni esplicite nell'esempio di prima:

```
class Stack {
        int top;
        int [] stack;
        Stack () {
          top=0;
          stack = new int [10];
    }
}
```

- Adesso basta `new Stack()` per completare l'inizializzazione.

Accesso ai campi

- Notare quando uso un campo mi riferisco alla *mia* particolare istanza:

```
class Num {
    int n;
    Num (int num) {
        n = num;
    }
}
```

- Creiamo due oggetti:

```
Num a = new Num(1);
Num b = new Num(2);
```

- Abbiamo che la n di a (`a.n`) vale 1, mentre la n di b (`b.n`) vale due, e sono separate e distinte.
- I costruttori (e i metodi) vengono invocati in un determinato "ambiente"
  - ♦ questo ambiente è l'oggetto corrente
- modificare i campi significa modificare il proprio oggetto lasciando inalterati gli altri
- L'oggetto corrente viene acceduto tramite this

```
Num (int num)  {
     n = num;
}
```

- equivale a:

```
Num(int num) {
    this.n = num;
}
```

- `Notare che n` è campo (persistente) mentre num è un parametro (non persistente) .

Allocazione

```
Stack s = new Stack ();
```

- In realtà fa due cose:
  - ♦ `new` alloca la memoria necessaria pari alla dimensione dell'oggetto
  - ♦ `Stack` è un costruttore chiamato da new dopo l'allocazione per inizializzare l'oggetto
- Se non specifico costruttori, viene definito automaticamente un "costruttore di default"

```
Stack() {}
```

- il costruttore di default non fa niente (in realtà, chiama i costruttori della classe base, ma lo vedremo poi)

Più costruttori

- Se ho un costruttore qualsiasi non ho quello di default

```
Stack (int n){ top=0; stack = new int [n];}
Stack s = new Stack (); //ERRORE!
```

- Posso avere vari costruttori:

```
Stack () { top=0; stack = new int [10]; }
```

- Posso concatenare i costruttori

```
class Stack {
      int top;
      int[] stack;
      Stack(int n) { top=0; stack = new int[n];}
      Stack(){ this(10);}
}
```

- `this(...)` deve essere il primo comando di un costruttore
  - ♦ la chiamata `this(...)` è cosa diversa dal puntatore `this.campo`

Inizializzazioni

```
Class Stack {
      int top=0; // espressione di inizializzazione
      int stack;
      Stack (int size){ stack = new int [size];}
      Stack (){ this (10);}
}
```

- Le espressioni di inizializzazione *appartengono a tutti i costruttori*
- Esse vengono eseguite **prima** del corpo di un costruttore ma dopo l'allocazione

- ♦ le espressioni di inizializzazione non possono contenere chiamate o comandi
- ♦ le espressioni di inizializzazione fanno parte di tutti i costruttori

- notare che `new int[size]`, richiedendo un parametro, va posto in un costruttore

Riferimenti in avanti

```
class Stack {
      Stack(){this(10);}
      Stack(int size){stack = new int [size];}
      int top=0;
      int stack;
}
```

- E' corretto! Riferimenti in avanti sono consentiti
- Prima vengono inizializzati i campi
- Poi viene eseguito il corpo costruttore

Ordine di inizializzazione

```
class Nums {
    int a=1;
    int b=a+1; // giusto
    int c=d; // sbagliato
    int d=0;
}
```

- Le inizializzazioni però vengono eseguite in ordine di apparizione

Metodi

In una classe posso dichiarare dei metodi:

```
class Stack {
 ...
    void push (int x){

        stack[top++]=x;
    }
    int pop() {
    return stack[--top];
  }
}
```

Uso di metodi

- I metodi possono essere richiamati solo avendo un oggetto (una *istanza* di una *classe* costruita invocando un *costruttore*).

```
Stack s = new Stack()
s.push(4);
Stack t = new Stack();
t.push(5);
```

- I riferimenti ai campi hanno sempre un"this"implicito

```
void push(int x) {
  stack[top++]=x;
}
```

equivale a:

```
void push(int x) {
     this.stack[this.top++]=x;
}
```

Overloading

- due o più metodi possono avere lo stesso nome
- i metodi devono avere argomenti diversi:

```
void move(int x, int y) {...}
void move(Point p){...}
```

- i metodi non si distinguono per il valore ritornato

```
void move(int x, int y){}
boolean move(int x, int y)  {}      // errore!
```

Differenza coi costruttori

- i metodi ritornano sempre un valore, anche void
- i costruttori no, e devono avere lo stesso nome della classe
- uno degli errori più frequenti:

```
    class Stack{
    void Stack(int x)  {} //no!!!
}
```

- questo è un metodo che si chiama come la classe, non un costruttore

Finalizzazione

- Abbiamo visto come creare un nuovo oggetto. In Java, a differenza di altri linguaggi, non è necessario distruggere gli oggetti.
- *Java rileva automaticamente gli oggetti inutilizzati*.
- Ogni oggetto non usato da nessuno viene riciclato automaticamente dalla "raccolto di spazzatura" (garbage collection) automatica.
- La garbage collection ricicla *la memoria* ma non altre risorse del sistema (per esempio file, windows, thread...).
- Per riciclare le risorse diverse dalla memoria, è possibile aggiungere un metodo `finalize()` ad ogni classe.
    - ♦ il metodo `finalize()` viene invocato una volta sola prima che venga effettuata la garbage collection.
- Si può richiedere esplicitamente la garbage collection utilizzando `System.gc()`

Si possono richiedere le finalizzazioni di tutti gli oggetti inutilizzati con `Runtime.runFinalization().`

Static e Final

- Le classi possono avere campi e metodi *static*
- Le classi possono avere campi e metodi *final*
- Le classi possono essere *final:*

cioè **non possono essere estese o derivate**

Campi statici

- I campi statici equivalgono alle variabili globali

```
class Stack {
```

```
    static int count =0;
    int top=0;
    int [] stack;
    Stack() {
        ++ count;
        stack = new int[10];
    }
}
```

- c'è un solo `count`, condiviso tra tutti gli oggetti
- occorre il nome della classe per accedere ad un campo statico

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

  - `out` è campo statico di `System`

Inizializzazione Statica

- un campo statico è una variabile condivisa tra le istanze di una classe
- i campi statici vengono inizializzati al caricamento della classe
- Esistono i blocchi static, eseguiti al caricamento della classe

```
class Quadrati {
   static int[] a =new int[10] ;
   static{
       for(int i=0; i<10; ++1)
           a[i]=i*i;
       }
}
```

Metodi statici

- equivalgono alle funzioni: occorre solo il nome della classe per accedere al metodo:

```
int n = Integer.parseInt("123");
```

- main è un metodo static:

```
public static void main(string[]arg s) {...}
```

Visibilità:

- dai metodi si vedono campi e metodi static e non static
- dai metodi static si vedono solo i campi e metodi static
  - campi e metodi static *non hanno* un oggetto corrente, e quindi non hanno un `this`.

```
class C {
    int x;
    static int y;
    int f() { y=1; }         //si
    static int  g() { x=1; } //no
}
```

Campi e metodi "final"

- un campo final non può essere modificato

```
final int max=1;
```

- costanti in java : campi static final

```
class Limit {
    static final int MAX=999;
    static final int MIN=0;
}
```

- usate con `Limit.MAX`, `Limit.MIN`

  notare che occorre sempre qualificare (cioè specificare la classe) anche le costanti
- metodi `final`: metodi non ridefinibili con l'ereditarietà (vedremo più avanti)

Package

- Ogni classe ha un "nome lungo" che specifica completamente il

package cui appartiene. Per esempio:

```
java.lang.String
java.util.Vector
java.io.InputStream
```

- è possibile abbreviare il nome lungo di una classe utilizzando

```
 import nome.package;
```

```
package hello;
import java.util.Vector;
import java.lang.*;

class Hello {
     void hello () {
        String s = "hello"; // invece di java.lang.String
        Vector v = new Vector(); // invece di java.util.Vector
     }
}
```

Import

- in realtà: `import java.util.Vector;`

  dice di considerare `Vector` come una abbreviazione di `java.util.Vector`
- senza import

  `Vector v = new Vector();`
  dà "Class not found"
- In tal caso occorre scrivere esplicitamente

```
java util.Vector=new java.util.Vector();
```

- `import java.util.*;`

  importa tutte le classi del package
- con la import sono possibili collisioni: per esempio:

```
import java.util.*;
import java.sql.*;
// ambiguo: java.util.Date o java.sql.Date?
Date d = new Date (); // ERRORE
java.util.Date=new java.util.Date(); // OK
```

- Ultima cosa: è sempre implicito un

```
import java.lang.*;
```

quindi per *System*, *String*, *Thread*, etc non occorrono *import*.

Classpath

- un package corrisponde ad una directory
- una classe corrisponde ad un file

```
Package java.util.Vector equivale a:
    File DOS : java\util\Vector.class
    File UNIX: java/util/Vector.class
```

- le directory e i file vengono cercati nel CLASSPATH

se `CLASSPATH=c:\java;c:\java\lib`
allora `java.util.Vector` viene cercata come:

1. `c:\java\lib\java\util\Vector.class`
2. `c:\java\util\Vector.class`

- il package di default (senza dichiarazione di package) non ha una sottodirectory: le classi vengono cercate nelle sole directory del classpath e non nelle sottodirectory.

L'interprete

- l'interprete standard del JDK (`java`) *se non è definita la variabile di ambiente CLASSPATH* ha nel CLASSPATH le librerie standard e la directory corrente

```
c:>javac Hello.java
c:>java Hello
Hello!
```

- in generale non è così semplice! A volte la variabile di un ambiente CLASSPATH viene considerata, ma a volte no! (Dipende da implementazioni e versioni)
- In generale il modo più sicuro è specificare completamente il CLASSPATH.
  - ♦ lo switch `-classpath` (interpreti `java` e `jre`) serve a specificare l'intero CLASSPATH
  - ♦ lo swicth `-cp` (solo `jre`) serve ad aggiungere al CLASSPATH

Zip e Jar

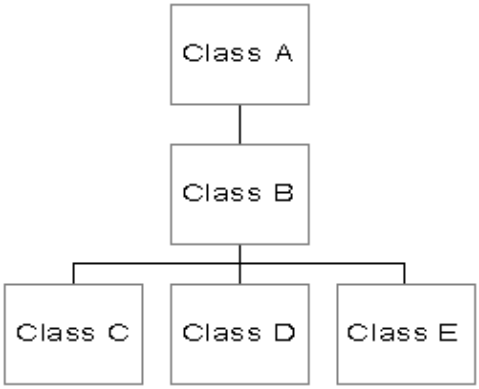
- nel CLASSPATH ci possono essere anche archivi in formato .zip o .jar, non compressi o (solo JDK1.1) compressi
- Tali archivi vengono trattati come directory, *sottodirectory comprese*
- Se `CLASSPATH=c:\java\lib\classes.zip;.`

`java.lang.String` è il file `String.class`.
che si trova nel `classes.zip` nella sottodirectory `java\lang`

Ereditarietà

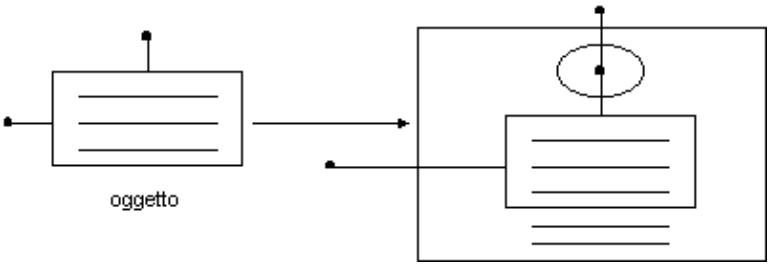
- L'ereditarietà consente di riutilizzare il codice esistente
- Il meccanismo di ereditarietà consente di riutilizzare i dati esistenti, aggiungendone di nuovi.
- Molto più importante è il meccanismo del polimorfismo, che consente di riutilizzare il codice esistente cambiandone il comportamento.

Gerarchia di classi



- Tramite l'ereditarietà si ottiene una gerarchia di classi.
- Una classe *estende* un'altra. La prima classe si chiama *superclasse* la seconda *sottoclasse* o classe derivata.
- A è la superclasse di B, che è la superclasse di C, D,E
- C D E sono derivate di B ma anche di A
- A e B sono superclassi di C, D ed E

L'idea

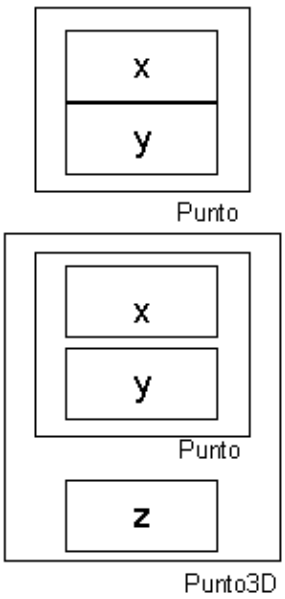


- Ereditarietà consente di generare nuovi clasi di oggetti che modificano quelli esistenti
- oggetto = stato(campi) + operatori(metodi)
- nuovo oggetto=

  stato esteso (nuovi campi) [ereditarietà]
  + metodi ridefiniti (nuovi metodi) [polimorfismo]
- Il nuovo oggetto estende lo stato aggiungendo informazione e ridefinisce (cambia) il comportamento dei metodi.

Esempi

Punto




```
class Punto {
   int x;
   int y;
}
class Punto3D extends Punto {
    int z;
}
```

- un punto 3D è un punto
- Punto p= new Punto3D();

Vediamo come estendere una classe `Pesce` e specializzarla in un `PesceRosso`.

```
class Pesce {
  int velocita = 10;
  void setVelocita(int x) { velocita = x; }
  void nuota() { ... }
}

class PesceRosso extends Pesce {
  int colore = "red";
  void mangia() { ... }
}
```

- La classe `PesceRosso` ha tutti i campi e i metodi di `Pesce` e in più un nuovo campo *colore* e un nuovo metodo *mangia()*.

Ridefinizione di metodi

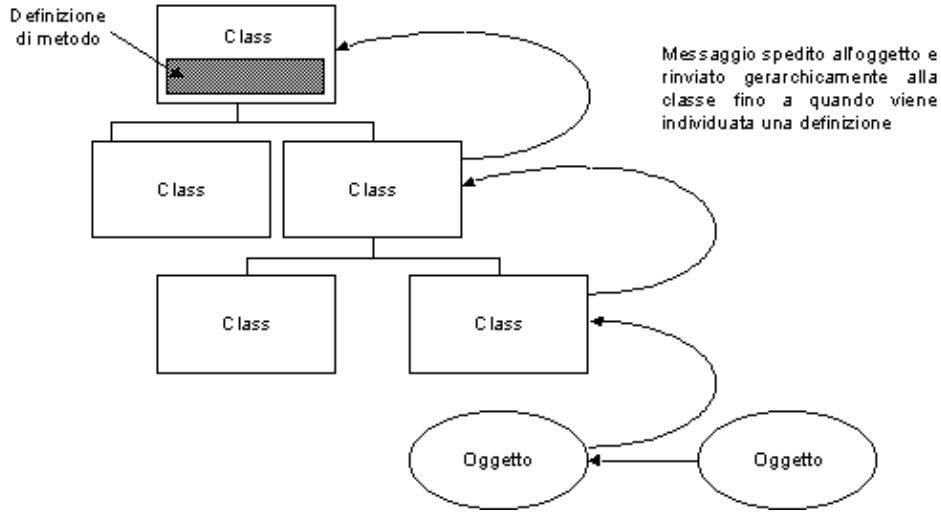
- un metodo con lo stesso nome ridefinisce un metodo della classe base

esempio:

```
class Component{
    boolean handleEvent (Event e){...}
}
class Window extends Component {
    boolean handleEvent(Event e){...}
}
```

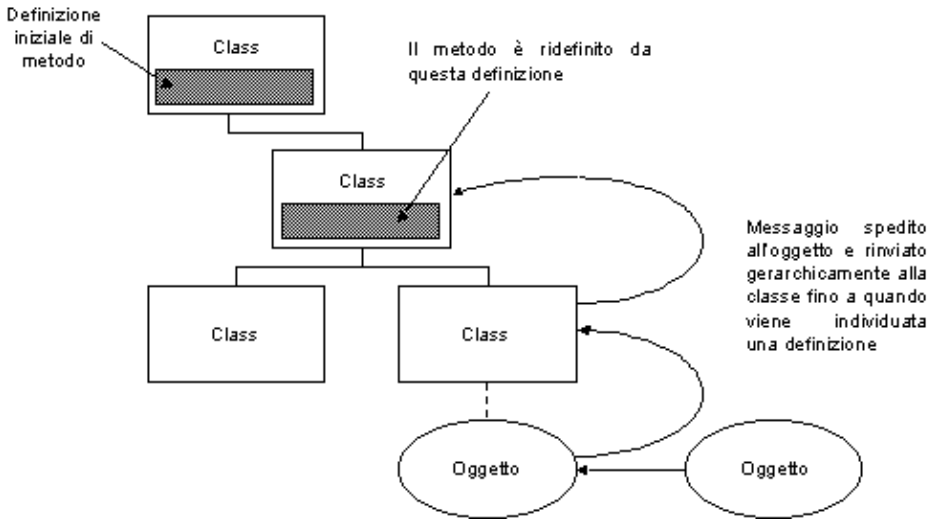
Invocazione di metodi

- Chiamando un metodo di una classe viene risalita la gerarchia finchè non si trova il metodo con il nome e il tipo di parametri corrispondenti alla chiamata.



- In questo modo un metodo può essere ridefinito.

Quando viene ridefinito verrà chiamato soltanto il nuovo.



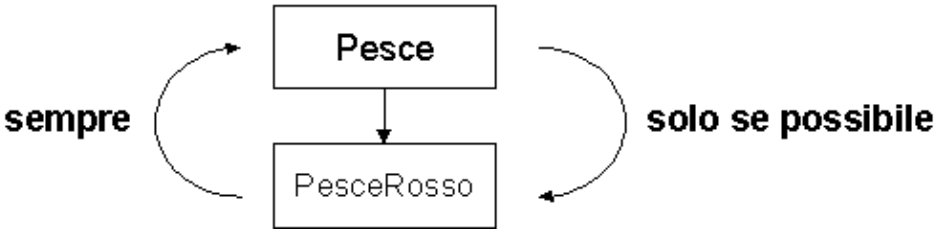
- Se occorre chiamare il metodo originario, si usa `super`

```
boolean handleEvent(Event e) {
    if (e.id==e.WINDOW_DESTROY)
        System.exit(0);
            else return super.handleEvent(e);
    }
}
```

- Chiamando un metodo con `super` *non* si applica il polimorfismo (vedi dopo), altrimenti si va in ciclo.

Conversioni

- una *PesceRosso* è **sempre** un *Pesce*
- ma un *Pesce* non è (in generale) un *PesceRosso*




```
Pesce p = new PesceRosso() ;
PesceRosso s =(PesceRosso) p;
```

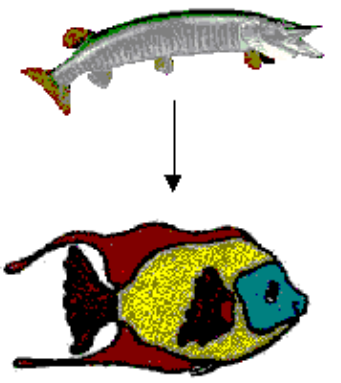
- la conversione è possibile solo se effettivamente un dato *Pesce* è in realtà un *PesceRosso*

Classe Object

- tutti gli oggetti estendono qualche oggetto
- Se non si estende un'altra classe, si estende (implicitamente) `Object`
- Object è l'unico oggetto che non estende nessun altro oggetto.
- Tutti gli oggetti ereditano (direttamente o indirettamente) da Object, per cui i suoi metodi sono presenti in tutti gli oggetti Java.
- Ogni oggetto può essere convertito ad `Object`, e infatti quando occorre riferirsi a *qualsiasi* oggetto si utilizza un riferimento ad `Object`.

`equals(Object)`	Confronto di oggetti
`hashCode()`	Codice HASH di un oggetto
`toString()`	Conversione in stringa
`clone()`	Duplicazione di un oggetto
`getClass()`	La classe di un oggetto
`finalize()`	Finalizzazione

Polimorfismo



- Supponiamo di avere un Pesce e un PesceRosso, entrambi con un metodo nuota..

```
Pesce.nuota()
```

```
PesceRosso.nuota();
```

- Posso assegnare un *PesceRosso* ad un *Pesce*

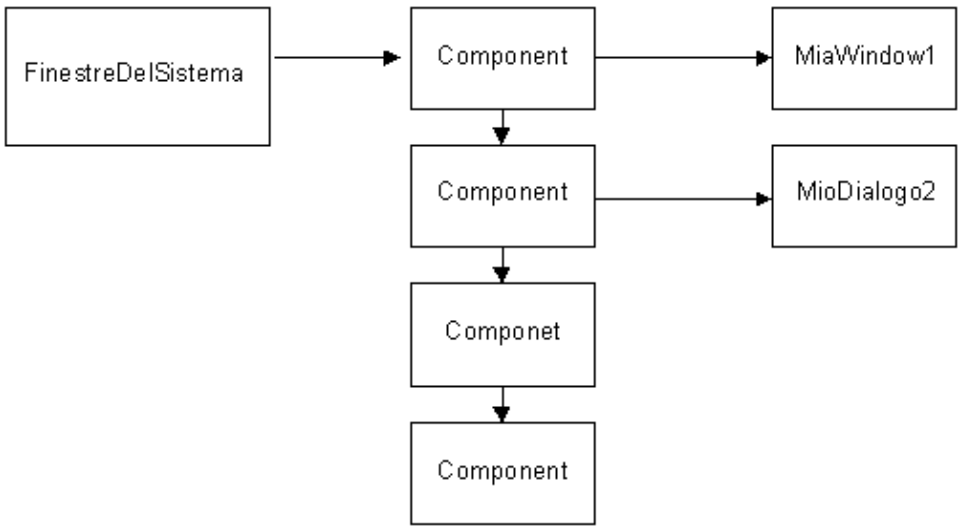
```
Pesce p= new PesceRosso();
```

- **Attenzione:** Se eseguo `p.nuota()` viene chiamato il metodo: `PesceRosso.nuota()`, *non* (come molti si aspetterebbero) `Pesce.nuota();`
- *Polimorfismo:ogni oggetto risponde in autonomia ai messaggi (le chiamate di metodi)* in base a quello che è, non in base a come viene considerato.
- Questo meccanismo è alla base del funzionamento dell'acquario.

```
class Acquario
{
    // un array di nuotatori
    Pesce[] elements;
    // aggiungo un pesce
    void add(Pesce n) {...}
    // posso scrivere codice anche senza averlo mai implementato...
    void refresh() {
    ...
        elements[i].nuota();
    }
}
...
Acquario a = new Acquario();
a.add(new PesceRosso());
a.add(new PesceSpazzino());
a.refresh();
// i pesci aggiunti nuoteranno a loro modo (come PesceRosso etc)
// anche se l'acquario li tratta come  dei Pesci qualsiasi.
```

Finestre

- Questo meccanismo è anche alla base della gestione di eventi della AWT (1.0), la libreria per GUI di Java
- Il sistema conosce solo che esistono dei `Component` a cui bisogna smistare gli eventi, chiamando `handleEvent`
- Il programmatore gestisce gli eventi ereditando dai componenti e ridefinendo `handleEvent`
- Grazie al poliformismo, l'evento viene smistato alla handleEvent definita dal programmatore.
- Notare che la AWT *non* può avere idea di come il programmatore gestirà gli eventi, e deve comunque prevedere che un componente possa ricevere eventi. Grazie al polimorfismo si semplifica molto la gestione, che in C viene complicato parecchio dalla necessità di avere dei callback.



Costruttori

- I costruttori non si ereditano: vanno ridichiarati, *uno per uno*
- Se non ho un costruttore viene implicitamente dichiarato il costruttore di default, ovvero il costruttore senza argomenti con un corpo vuoto.
- Ogni costruttore *per prima cosa* costruisce la classe base, chiamando uno dei costruttori della classe base. Quindi il primo comando di un costruttore deve essere `super(...)`, con o senza argomenti.
- Se il programmatore non inserisce una chiamata `super(...)`, ne viene aggiunta una implicitamente.
- L'unica eccezione a questa regola è quando viene utilizzato `this(...)` come primo comando di un costruttore. In tal caso non viene chiamato `super(...)`, ma viene eseguito un altro costruttore il quale presumibilmente chiamerà `super(...)` o passerà la palla ad un altro costruttore finchè qualcuno chiamerà una `super(...)` (altrimenti si va in ciclo!).
- Esempio:

```
class B{
      B() {...}
      B(int x) {...}
}
class D extends B{
    D(){super(0); }       //chiama B(int)
    D(int x) {this.x=x;}  //chiama B ()
    D(String s) {
      this(Integer.parseInt(s));...
    }
 }    //chiama D(int)
     // che chiama B()
```

Ordine di costruzione

Gli oggetti vengono costruiti in questo ordine:

1. Prima la classe base
2. Poi vengono inizializzati i campi
3. Infine si esegue il costruttore

- Esempio:

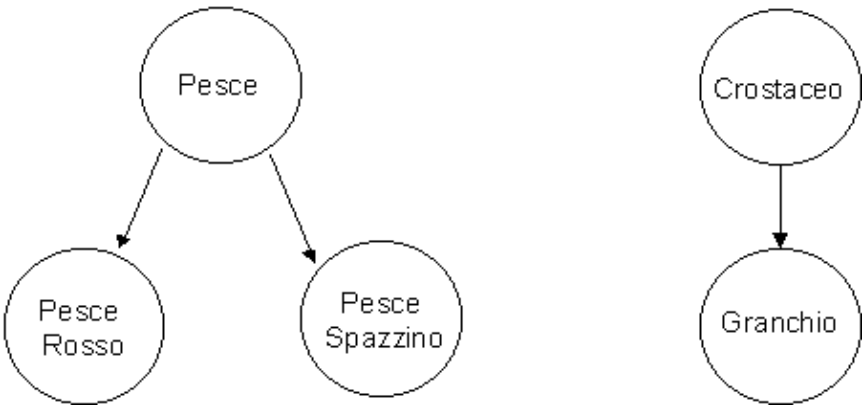
```
class D extends B{
  int x=1;
  int y;
  D() {
    super(1);
    y=2;
  }
}
```

- Ordine:

1. Prima viene chiamato il costruttore della classe base:
   `super(1)`
2. Poi viene inizializzato il campo:
   `int x=1;`
3. Infine viene eseguito il corpo del costruttore
   `y=2;`

Interfacce

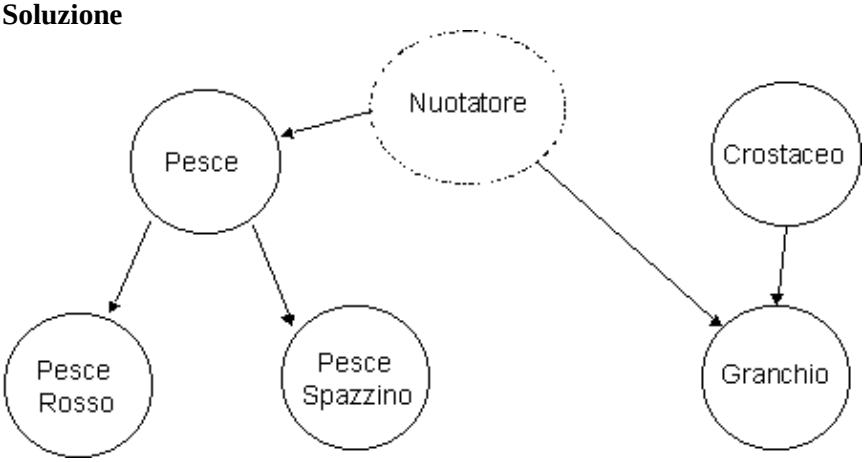
Il Problema



- Vogliamo inserire in un Acquario un Granchio
- Un Acquario vuole solo Pesci
- Non possiamo rendere un Granchio anche Pesce perchè Java fornisce solo l'ereditarietà singola (e Grancho è già un Crostaceo)

Che facciamo?

Soluzione



- Definiamo una interfaccia `Nuotatore`
- Le interfacce sono classi "vuote"
- Mettiamo nell'acquario dei Nuotatori, anzichè Pesci
- Java supporta l'ereditarietà multipla di interfacce

Dichiarazione di interfaccie

Esempio di dichiarazione di interfaccia:

```
interface Nuotatore{
    public void nuota();
    public static final int SLOW=0;
    public static final int FAST= 1;
}
```

- Una interfaccia ha solo le"signature" (dichiarazioni) dei metodi
- Una interfaccia ha solo campi *static final* (costanti)

Per dichiarare che una classe supporta una interfaccia (si comporta come dichirato dall'interfaccia) occorre

"implementarla". Se un programmatore implementa una interfaccia è anche obbligato a definire tutti i metodi dichiarati nell'interfaccia.

```
class Pesce implements Nuotatore {
    public void nuota(){...}
}
class Granchio implements Nuotatore {
    public void nuota(){...}
}
```

Uso di interfaccie

- Non è possibile costruire oggetti di un tipo interfaccia

```
new Nuotatore();   //no!
```

- E' però possibile dichiarare (riferimenti a) oggetti di un tipo interfaccia:

```
Nuotatore p = new PesceRosso();
Nuotatore g = new Granchio();
```

- E' quindi possibile chiamare i metodi di una interfaccia:

```
p.nuota();
g.nuota();
```

- L'interfaccia è un *modello* del comportamento di un oggetto
- Si pùo programmare senza nulla conoscere della effettiva implementazione

```
class Acquario
{
    // un array di nuotatori
    Nuotatore[] elements;
    // aggiungo un nuotatore
    // − quindi devo avere un nuotatore per aggiungerlo!
    void add(Nuotatore n) {...}
    // posso scrivere codice anche senza averlo mai implementato...
    void refresh() {
    ...
        elements[i].nuota();
    }
}
```

Visibilità

- Java dispone di modificatori che alterano la visibilità dei membri e classi
- Una classe può essere `public` o default (nessun modificatore):
  - Una classe `public` è visibile da altri package. **In tal caso la classe deve stare in un file con il suo stesso nome.**
  - Una classe senza modificatori è visibile solo dal altre classi nello stesso package.
- Un membro di una classe (metodo o campo) può avere visibilità `private`, `protected`, default (nessun modificatore) o `public`.
  - `Private` è visibile soltanto all'interno della classe
  - `Protected` è visibile soltanto da classi derivate
  - Default è visibile soltanto all'interno del package
  - `Public` è visible anche all'esterno del package.
- Riepilogo tabellare:

Accessibile a:	Visibilità del Membro			
	pubblico	protetto	package	privato
Stessa classe	si	si	si	si
Classe nello stesso package	si	si	si	no
Sotto-classe in differente package	si	si	no	no
Non-sotto-classe, differente package	si	no	no	no

Eccezioni

Cosa sono:

```
f() { ... g()...}
g() {... h()... }
h() {... ECCEZIONE!...}
```

- Abbiamo f() che chiama g() che chiama h(). A un certo punto in h() scatta una eccezione.
- Se l'eccezione non viene gestita, *viene causato un return e dal punto di chiamata viene sollevata una eccezione*.
- Questo meccanismo viene iterato: in h() si ritorna in g(), che causa un ritorno in f() che a sua volta causa un ritorno al chiamante di f().
- Se nessuno gestisce l'eccezione alla fine il programma termina e l'eccezione viene intercettata dalle librerie di sistema che stampano un messaggio di errore.
-

Throw

- Una eccezione (normale) è un oggetto derivato da `Exception`
- Per convenzione le eccezioni hanno un nome che termina in `Exception`
- Come sollevare una eccezione:

```
throw new Exception ("disastro!");
```

- Una eccezione è un valore ritornato, quindi se viene sollevata una eccezione bisogna dichiararlo:

```
void f() throws Exception {
    ...
   throw new Exception( );
   ...
}
```

- Il compilatore si accorge delle eccezioni contenute in un programma, perchè esamina i throw e le dichiarazioni dei metodi, e quindi è in grado di stabilire se il metodo è dichiarato correttamente.

```
void g ( )  throws Exception { ... }
// poichè f() contiene g() può sollevare eccezioni e va dichiarato
void f ( )  throws Exception { ... g(); ... }
```

- Le eccezioni, o si propagano o si gestiscono
- Per ignorare una eccezione:

```
try {
```

```
  f();
} catch (Exception ex) {
  ex.printStackTrace( );
}
```

- A volte non si può propagare una eventuale eccezione, occorre gestirla ignorandola.
- In ogni caso è meglio lasciare *sempre* traccia di una eccezione ignorata stampando qualcosa: può essere utilissimo per scoprire errori inaspettati.

Sintassi

La sintassi completa della gestione eccezioni è:

```
try {
  ...
} catch (MiaEccezione ex) {
  ...
}catch (Exception e) {
  ...
} finally {
 ...
}
```

try

- Raffiniamo passo passo un esempio che mostra un programma che copia un file.
- Gestione "grossolana": si cattura l'eccezione , si termina e si stampa il trace della stessa.

```
try {
    String fin = args[0];
    String fout= args[1];
    InputStream in
        = new FileInputStream(fin);
    OutputStream out
        = new FileOutputStream(fout);
    int c;
    while( (c=in.read()) != -1)
        out.write(c);
    in.close();
    out.close();
} catch (Exception ex) {
    ex.printStackTrace();
}
```

Notare che ci sono diverse eccezioni che possono essere sollevate:

- `args[0]`
- `throws ArrayIndexOutOfBoundsException`
- `FilelnputStream(String)`
- `throws FileNotFoundException`
- `in.read()`
- `throws IOException`

catch

```
try {
  String fin = args[0];
  ...
  InputStream in = new FileInputStream(fin);
  ...
} catch(ArrayIndexOutOfBoundsException ex) {
    System.out.println("bad args");
} catch(FileNotFoundException ex) {
    System.out.println("file not found");
}
```

- Possiamo reffinare intercettando le eccezioni di vario tipo e comportandoci di conseguenza
- Si deve inserire una catch() con un parametro di tipo compatibile (cioè uguale o derivato) all'eccezione che si vule intercettare
- Inserendo una catch(Exception ex) si intercettano tutte le eccezioni normali.Ci sono altri eccezioni non derivate da `Exception`, (derivate da `RunTimeException` e da `Error`) che vengono generate dal sistema in casi particolari e che *non* dovrebbero essere normalmente utilizzate.
-

- Le eccezioni vengono provate in ordine, quindi bisogna ordinare le catch() in modo che le eccezioni più generali stiano in fondo (altrimenti certe catch sarebbero irrangiungibili in quanto le catch più generali "catturano tutto").

```
try {
 InputStream in =new FileInputStream (fin);
  ...
  c=in.read();
  ...
} catch(FileNotFoundException ex) { // 1
  ...
}catch(IOException ex) {         // 2
 ...
}
```

- Scambiando 1 e 2, la 1 non viene più raggiunta perchè
- `FileNotFoundException` deriva da `IOException` e quindi cattura e propaga la prima.

Propagazione

- Se nessuna catch cattura l'eccezione, questa viene propagata
- Una catch può catturare l'eccezione, esaminarla e risollevarla

```
catch(SpecialException ex) {
   if(<non-gestibile)
        throw ex;
}
```

- Una catch può trasformare l'eccezione

```
catch(SpecialException ex) {
    if(<non-gestibile)
        throw new OtherException (ex);
}
```

- Trasformazione utile con l'ereditarietà

finally

```
try{
  ...
} catch (...) {
  ...
} catch(...) {
  ...
} finally {
 //sempre
}
```

- Il blocco *finally* viene eseguito in qualunque modo il controllo lasci il blocco *try*...
- ♦ Sia che la *try* completi normalmente
  ♦ Sia che venga sollevata una eccezione e intercettata da una *catch*
  ♦ Sia che venga eseguito un *return*

Il blocco `finally` viene utilizzato per effettuare pulizie finali.

```
InputStream in=null;
OutStream=null;
try{
  ...
  in= new FileInputStream (fin)
  out=new FileOutputStream (fout)
} catch (...) {
  ...
}finally {
  ...
  if(in!=null)
   in.close();
  if(out!=null)
  out.close( );
}
```

- Il file viene chiuso se è stato aperto
- Attenzione che anche `close()` solleva eccezioni!

Esempio

Ecco l'esempio completo:

```
public class Eccezioni {
    public static void main(String [] args) {
        InputStream in=null;
        OutputStream out=null;
        try {
            int c;
            in=new FileInputStream (args [0] );
            out=new FileOutputStream (args [1];)
            while( (c=in.read( )) != -1)
                out.write (c);
        }catch(ArrayIndexOutOfBoundsException ex) {
            System.out.println ("not enough args");
        }catch(FileNotFoundException ex) {
            System.out.println ("file not found");
        }catch (IOException ex) {
            System.out.println ("I/O error");
```

```
        }finally{
            try {
                if (in!=null)
                    in.close( );
                if( out!=null)
                    out.close( );
            } catch(Exception ex) {
                ex. printStackTrace ( );
            }
        }
    }
}
```

Ereditarietà

Esaminiamo la relazione tra le eccezioni e l'ereditarietà

```
class Pesce {
    void nuota()
        throws PesceMortoException { ... }
}
class PesceAtomico extends Pesce {
    void nuota()
        throws PesceMortoException,
               EsplosioneException { ...  } // NO!
}
```

- I metodi ridefiniti non possono aggiungere nuove eccezioni
- Altrimenti codice prima funzionante non funzionerebbe più:

```
class Acquario {
    ...
    try {
      p.nuota() ;
    }catch (PesceMortoException pme) {
    ...
   }
}
// se aggiungo un PesceAtomico, il nuovo pesce può sollevare nuove eccezioni imprevedibili
// vanificando il codice che lo intercetta e lo gestisce
// inoltre non si potrebbe determinare quali eccezioni realmente sono sollevabili
```

- Possiamo però aggiungere nuove eccezioni *derivando* una eccezione da quelle esistenti.
- In questo modo il codice esistente e le dichiarazioni dei metodi rimangono valide
- Si può quindi riconvertire le eccezioni "anomale" in eccezioni compatibilie.

```
class PesceMortoPerEsplosioneException extends PesceMortoException { }
class PesceAtomico extends Pesce {
    void nuota()
        throws PesceMortoPerEsplosioneException
     {
        try {
            <operazione-a-rischio-di-esplosione>;
        } catch (EsplosioneException ex) {
            throw new
               PesceMortoEsplosioneException ( ) ;
        }
    }
}
```

Input/Output

Consideriamo i concetti principali della package per la gestione di Input/Output

- Il package per l'I/O è java.io, e le classi si usano con `import java.io.*`
- Il concetto principare è lo Stream, un oggetto in cui si scrive (`OutputStream`) e da cui si legge (`InputStream`)
- Ci sono due gruppi di classi molto simili, un gruppo per l'Input ed l'altro per Output
- Ci sono casi particolari come `RandomAccessFile`
- Gestione di nomi di file indipendente dalla piattaforma
- Tutte le eccezioni di I/O sono derivate da `IOException`

Complicazioni

- L'unità di I/O in Java è il *byte* (a 8 bit) non il *char* (a 16 bit), che necessitano di gestione particolare.
- Le Stringhe sono sequenze di caratteri Java a 16 caratteri, quindi l'I/O utilizza delle codifiche. Per esempio l'UTF che consente di esprimere una stringa unicode con caratteri ASCII
  - L'Unicode è un superset dell'ASCII, e l'UTF (che codifica i caratteri non ASCII con sequenze tipo \uABCDE) è efficiente quando la maggior parte dei caratteri sono ASCII
- L'I/O di caratteri avviene altrimenti per troncamento/estensione
- scrittura di caratteri a 16bit: troncati in byte 8bit (si scarta il byte alto)
  lettura di byte a 8bit: estesi a char 16bit (byte alto nullo)

InputStream

InputStream	
`int read()` `  throws IOException`	Ritorna un byte, da 0 a 255 oppure −1 che significa fine−input E' bloccante: si aspetta finchè non è disponibile un carattere

`int read(byte[] buf, [int off,`
`int  len])`
`  throws IOException`
	legge fino buf.length caratteri
	ritorna il numero di byte letti oppure −1 (EOF)
	bloccante

`long skip (long count)`	salta count caratteri (o fino a EOF) ritorna il numero di byte saltati bloccante
`int available()`	quanti byte posso leggere senza bloccarmi
`void close()`	`chiude lo stream` `comumque lo fa il finalize()`

OutputStream

OutputStream	
`void write(int b)` `  throws IOException`	vengono scritti gli 8bit più bassi è int per evitare un cast

	bloccante
`void write (byte[] buf [,int offset, int count]) throws IOException`	scrive buf.lenght caratteri bloccante
`void flush( )`	svuota il buffer
`void close( )`	chiude lo stream

File

- Classe wrapper per operazioni legate ai file
- Un oggetto `File` rappresenta il nome di un file, non l'intero file
- Gestisce anche varie operazioni come relative alle directory

File	
`File (String path)` `File (File dir, String name)`	Costruisce un file a partire da un path o da un File che rappresenta una directory e un nome di file.
`long length ( )` `long lastModified ( )`	Ritorna la lunghezza o la data di ultima modifica in termini di millisecondi dal 1/1/1970
`boolean exist()` `boolean canRead()` `boolean canWrite()`	Ritorna se un file esiste, è leggibile o scrivibile
`boolean isFile()` `boolena isDirectory()` `boolean isAbsolute()`	E' un file , una directory, il path è assoluto?
`boolean mkdir()` `boolean mkdirs()`	Crea una directory / tutte le directory intermedie in un path
`boolean renameTo(File newName)` `boolean delete()`	Rinomina o cancella un file
`String getName()` `String getPath()` `String getParent()` `String getAbsolutePath()`	Estrae il nome, il path, la directory padre o il path assoluto.
`String[] list()` `String[] list(FilenameFilter filter)`	Lista i file di una directory
`long getFD`	Ritorna un file descriptor

FileStream

- `FileInputStream` e `FileOutputStream` sono stream per leggere e scrivere nei file
- Costruttori:
- `FileInputStream(String)`
  costruisce da un nome di file (path)
  `FileInputStream(File)`
  costruisce da un file
  `FileInputStream(FileDescriptor)`
  costruisce da un descrittore di file
- `FileOutputStream`
- è analogo

DataStream

- Sono stream filtro, nel senso che operano su uno stream esistente e vi aggiungono i metodi per potenziare le capacità degli stream base
- `DataInputStream` e `DataOutputStream` consentono di leggere e scrivere i vari tipi di Java (non solo byte):
- `boolean, char, byte, short, int, long, float, double`
  con metodi dal nome ovvio: `readBoolean, writeBoolean`...
- Notare la `readLine()` per leggere una linea di testo
- che riconosce gli End−Of−Line (\n, \r, \n, \r, o \r\n)
  ritorna null quando è fine−file
- Non esiste la corrispondente `writeLine()`
- Invece esiste `writeBytes (String s)` (che taglia il byte alto)

Classi varie

- `RandomAccessFile`
- consente la lettura e la scrittura in qualunque posizione di un file
- `BufferedInputStream, BufferedOutputStream`
- stream bufferizzati
- `PushbackInputStream`
- stream filtro con "`unread (int)`" per mandare indietro un byte
- `PrintStream`
  print e println per: String, char, int., long...
  con autoflush quando incontro un '\n' (disabilitabile)
  `System.out` è un `PrintStream`
- `SequenceInputStream` concatena più Stream:

```
FileInputStream files = {
    new FileInputStream  ("a"),
    new FileInputStream ("b")  };
InputStream in =SequenceInputStream (files);
```

- `PipedStream`
  collega uno stream di input in uno stream di output
  `PipedOutputStream out= new PipedOutputStream() ;`
  Ci vogliono due Thread per usarlo, uno che scrive e uno che legge.
- ByteArrayInputStream e ByteArrayOutputStream
  sono Stream "costanti"da usare per leggere dati fissi o per scrivere in memoria dati

- ♦ `ByteArrayInputStream(byte[] data)`
  - ♦ Costruisce un input stream costante da un array di byte
  - ♦ `ByteArrayOutputStream ()`
  - ♦ Costruisce un output stream che raccoglie byte
    - ◊ `size()`
    - ◊ Numero di byte scritti
    - ◊ `byte[] toByteArray()`
    - ◊ Restituisce i byte scritti come un array
    - ◊ `String toString ([int hibyte])`
    - ◊ Restituisce i byte scritti come una stringa, usa hibyte come byte alto (0 default)

Stringhe

`String()` crea una stringa vuota
`String(String)` crea una stringa copia di un'altra
`int length()` ritorna la lunghezza
`char charAt(int)` ritorna l'i−simo carattere

Esempi

// Codice Cesare: a=d, b=e, etc

```
String s = "hello";
for(int i=0; i<s.length(); ++i)
System.out.print(s.charAt(i)+3);
```

Ricerche

```
int indexOf(char c [, int start])
int indexOf(String s [, int start])
int lastIndexOf(char c [, int start])
int lastIndexOf(String s [, int start])
```

ricerca la prima/ultima posizione del carattere/stringa [a partire dalla posizione start]; ritorna la posizione o −1

// Conteggio di asterischi

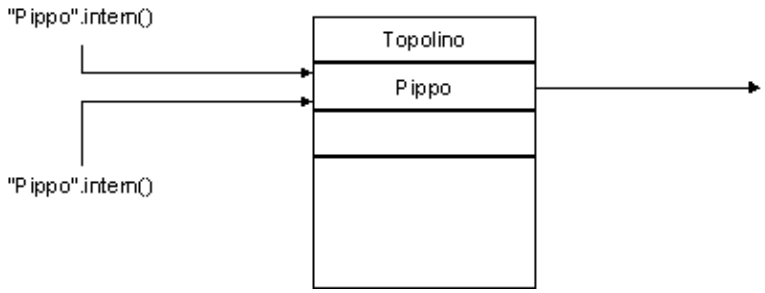
```
int count = 0;
for(int curr= s.indexOf('*');
    curr != −1;
    curr = s.indexOf('*', curr))
         ++count;
```

Confronto

In generale, non si può fare un confronto s==t (per fare questo occorre l'internamento)
`boolean s.equals(t)`
`s.equalsIgnoreCase(t)` vale anche per le lettere accentate: à e À
`int s.compareTo(t)` ritorna −1 se s<t, 0 se s=t, 1 se s>t
`boolean s.startsWith(String prefix [, int offset])`
`boolean s.endsWith(String suffix)` verifica se combaciano prefix/suffix [a partire da offset]

```
boolean s.regionMatches([boolean ignoreCase,] int start, int other,  int
ostart, int len)
```

In generale due stringhe anche con lo stesso contenuto sono due oggetti diversi.
`String s = "pippo" ; String t = "pippo"; ma s!=t` (puntano a oggetti diversi).
E' possibile fare in modo che se s==t allora `s.equals(t)`
Importante per ragioni di efficienza
`s.intern()` restituisce un puntatore ad una stringa tale che se `s.equals(t)` allora `s.intern() == t.intern()`
Implementazione tramite una tabella:



Utility

```
String replace(char oldChar, char newChar)
void toLowerCase() String.toUpperCase()
String trim()
String substring(int begin [, int end])
```

Attenzione: inizio incluso, fine esclusa

StringBuffer

Ci sono due classi che rappresentano le stringhe: `String` e `StringBuffer`
String una volta costruita è immutabile: non è possibile cambiare il contenuto nè ridimensionarla. String comunque è ragionevolmente efficiente.
`StringBuffer` è invece una stringa "mutabile":
`setCharAt(int index, char ch);`
`append(String str)`
`insert(int index, String s)`
`StringBuffer` ha una lunghezza (`length()`) e una capacità (`capacity()`)
La capacità può essere assicurata con `ensureCapacity()`

```
Stringhe, byte[] e char[]
Da byte[] e char[] a String
String(char[], [int offset, int count])
String(byte[], [int offset, int count], int hiByte)
```

Viceversa:

```
getBytes(int srcBeg, int srcEnd, byte[] dst, int dstBeg)
getChars(int srcBeg, int srcEnd, byte[] dst, int dstBeg)
```

StringTokenizer

La `java.util.StringTokenizer` suddivide in token una stringa. Molto utile per il parsing (frequente) di input testuale in formato delimitato

```
StringTokenizer st
    = new StringTokenizer("this is a test");
while (st.hasMoreTokens()) {
   println(st.nextToken());

}
```

```
this
is
a
test
```

`StringTokenizer(String str, [[String delim], boolean returnTokens])` dove delim sono i delimitatori che di default sono sequenze di "\r\n\t ", mentre returnTokens specifica se il delimitatore fa parte del token

Contenitori

Il package `java.util.*` contiene molte classi contenitore:

- Vector: simili agli array ma espandibili
- Hashtable: tabelle chiave/valore
- Enumeration: enumerazione di elementi contenuti in un contenitore

Enumeration

`java.util.Enumeration`: interfaccia standard per enumerare i contenuti di qualsiasi contenitore: (vettori, tabelle, liste…)

Usa i metodi:

```
boolean hasMoreElements()
Object nextElement()
```

Esempio:

```
Enumeration e = table.elements();
while(e.hasMoreElements())
  doSomething(e.nextElement());
```

Vector

`java.util.`Vector è un array dinamico
con dimensione e capacità (ovvero elementi inseribili senza espandersi)

`Object elementAt(int i)` ritorna l'i−simo elemento
`void setElementAt(int i, Object o)` modifica l'i−simo elemento

Vector contiene soltanto Object;
l' accesso agli elementi avviene tramite `Object elementAt(int i)` che ritorna l'i−simo elemento
`void setElementAt(int i, Object o)` imposta l'i−simo elemento

Rispetto agli array, java.util.Vector può essere ridimensionato

`void addElement(Object o)` aggiunge in coda
`void insertElementAt(Object o, index i)` aggiunge dopo l'i−simo
`void removeElementAt(index i)` rimuove l'i−simo elemento

java.util.Vector effettua ricerche: `int indexOf(Object o [,int index])` trova l'oggetto
`int lastIndexOf(Object o [,int index])` trova l'oggetto dalla fine boolean
`removeElementAt(Object o)` rimuove l'oggetto

Dictionary

La `java.util.Dictionary` è una interfaccia che rappresenta una tabella di associazioni

Ogni associazione ha un oggetto chiave e un oggetto valore
`void put(Object k, Object v)` aggiunge associazione chiave/valore (se a k è già associato un altro v l'associazione viene persa)

`Object get(Object k)` ritorna l'oggetto v associato correntemente a k
`void remove(Object k)` rimuove l'associazione con chiave k

Altre operazioni:

`boolean isEmpty()` mi dice se è vuoto
`void size()` mi indica il numero di associazioni
`Enumeration keys()` mi da l'enumerazione delle chiavi
`Enumeration elements()` mi da l'enumerazione dei valori

Hashtable

La java.util.Hashtable implementa il Dictionary utilizzando la tecnica delle tabelle hash
Ci sono sempre get, put, elements, etc.

La `java.util.Hashtable` utilizza `Object.hashCode()` e `Object.equals(Object)`

Ha una capacità e una load factor: se `size()*loadFactor > capacity()` la capacità viene incrementata.

`boolean contains(Object)` contiene il valore boolean `containsKey(Object)` contiene la chiave

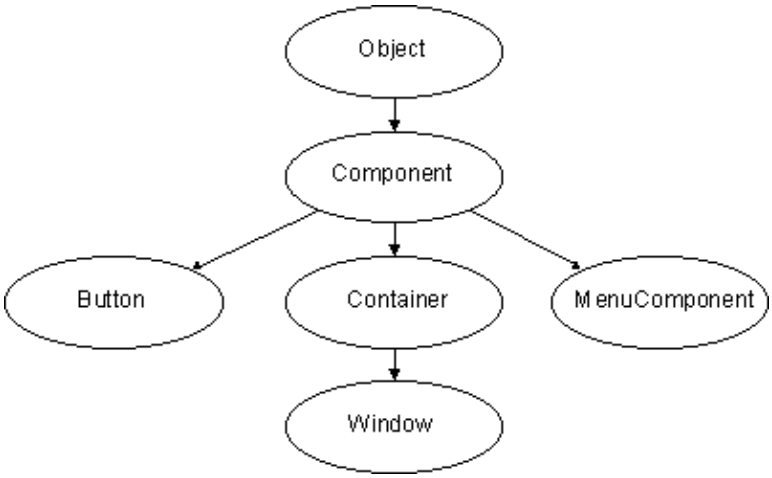
Random

La `Random([long seed])` inizializza il generatore di numeri pseudocasuali con dato seme, default l'ora corrente.
`setSeed(long)` cambia il seme
`nextInt()` genera valore intero casuale
`nextLong()` genera valore long casuale

`nextFloat()` genera valore float casuale
`nextDouble()` genera valore double casuale
`nextBytes(byte[] bytes)` genera array di byte casuale

AWT

Il package AWT, può essere così schematizzato:



Componenti e Container

- Alla base dell'AWT ci sono i `Component`, che hanno la proprietà di essere sensibili agli eventi.
- Derivati dai componenti, ci sono i `Container`, che sono in grado di contenere i componenti.
- Molto importante: i `Container` sono essi stessi dei `Component` e quindi un `Container` può contenere un altro `Container`.
- L'annidamento dei container è basilare nella gestione delle interfacce in Java.

Panoramica

Container: Panel, ScrollPane, Window, Frame, Dialog
Component: Button, Canvas, Checkbox, Choice, Label, List, TextField, TextArea
LayoutManager: BorderLayout, GridLayout, FlowLayout, CardLayout, Menu: MenuBar, Menu, MenuComponent, MenuItem, CheckboxMenuItem, MenuShortcut
Gestione Eventi 1.0: Component.handleEvent(), Event
Gestione Eventi 1.1: WindowEvent WindowListener WindowAdapter

Una finestra

Vediamo ora come costruire una finestra usando AWT.

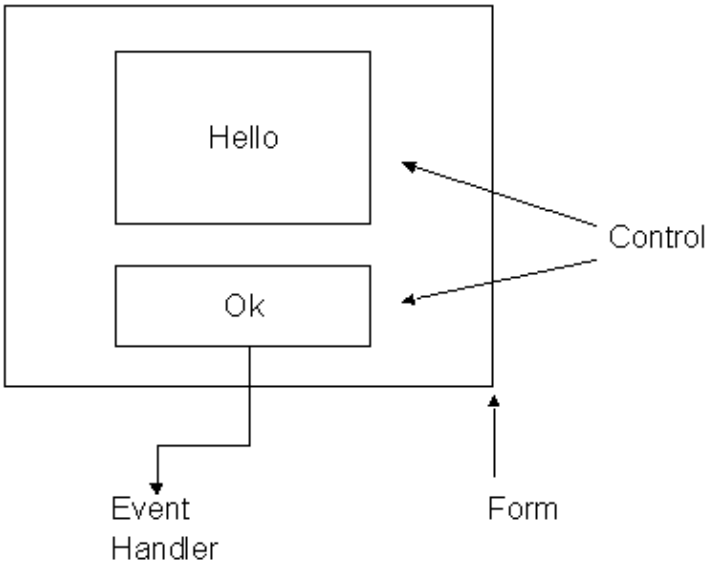
```
import java.awt.*;
import java.awt.event.*;
public class AWT1
      extends Frame
      implements ActionListener
{
```

```
   Label hello = new Label("Hello");
   Button ok = new Button("OK");
    AWT1() {
     add("Center", hello);
     add("South", ok);
     setSize(100,100);
     pack();
     ok.addActionListener(this);
     setVisible(true);
}
   public void actionPerformed(ActionEvent e) { System.exit(0); }
   public static void main(String[] args) {
     new AWT1();
   }
}
```

In questa finestra è presente una scritta al centro dentro un quadrato con scritto "hello", e un bottone in basso con su scritto "OK". Premendo il bottone non vi è associato alcun evento particolare.

Analogie col Visual Basic

Riportiamo in figura la finestra creata nell'esempio precedente.



In Visual Basic per creare una finestra simile si crea una form.
All'interno si disegnano i `Control:` campi e bottoni.
Per associare eventi ai bottoni basta fare doppio click e scrivere l'evento che si vuole associare.

In Java si dichiara una classe che estende la classe **Frame**

```
class AWT1 extends Frame {
```

All'interno si dichiarano e si costruiscono i `campi`: la label e il button

```
 Label hello = new Label(“Hello”);
```

```
 Button ok = new Button (“ok”);

}
```

Volendo fare un'analogia con il Visula Basic
La classe è la Form e i campi sono i Control della Form.

Vediamo in dettaglio i vari componenti.

Costruire una Form

```
class Form extends Frame {

      Form(){
          add(“Center”, hello);
          add(“Smith”, ok)
          …
      }
}
```

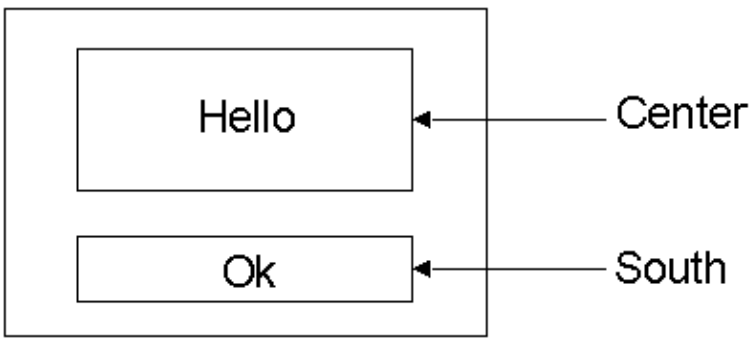
nel costruttore si posiziona i control nella form.
Notare che NON SONO USATE coordinate ma nomi simbolici. Infatti grazie al `Layout Manager` è possibile usare *nomi simbolici* anzichè coordinate *x, y*.

```
class Form extends Frame
implements ActionListener {
      Form(){
          ok.add ActionListener(this);
      }
      void actionPerformed (Action Event e){
            System.exit(0);
      }
}
```

Nel costruttore viene “registrato” il target dell’evento
il metodo actionPerformed viene chiamato per gestirlo

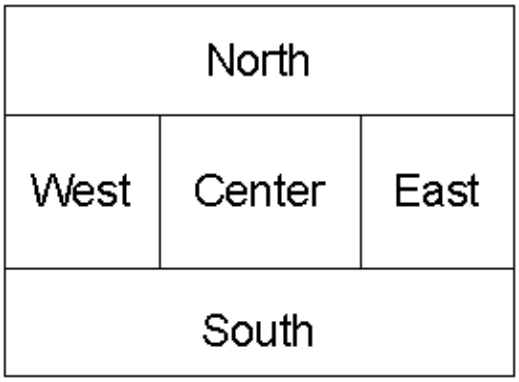
Layout Manager

I componenti vengono aggiunti geometricamente ad un contenitore:



Ogni contenitore ha associato un LayoutManager che gestisce il posizionamento di un componente.
Questo si imposta utilizzando Container.setLayout(LayoutManager).
Frame ha di default i seguenti "gestori": il BorderLayout, Panel e il FlowLayout

BorderLayout



Il `Border Layout` dispone i componenti a croce
Ogni componente ha un preferredSize x e y a seconda della posizione assunta.
E precisamente:

- il BorderLayout “allunga” x e y se posizionato al centro
- il BorderLayout “allunga” x ma pone y alla preferredSize a nord e sud se posizionato di lato nella parte centrale
- il BorderLayout “allunga” y ma pone x alla preferredSize a est e ovest se posizionato in basso o in alto

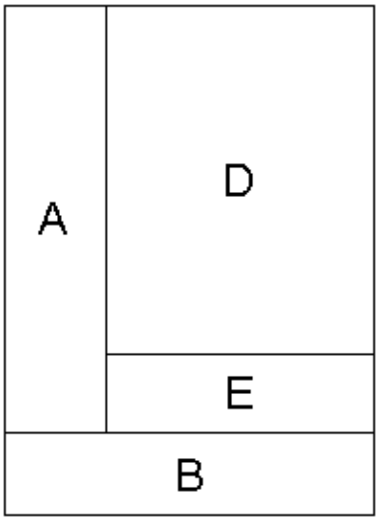
notare che il posizionamento avviene in maniera tale che PRIMA si estendono a nord e sud e POI a est e ovest.

Pannelli

Posso mettere soltanto UN componente a Nord, Sud, Est, Ovest.
Se si vuole fare posizionamenti diversi da questi c’è un contenitore che permette di raggruppare più componenti: il Panel

```
Panel p = new Panel();
p.setLayout(new BorderLayout())
p.add(“Center”, new Label(“D“));
p.add(“South”, new Label(“E“ ))
add(“Center”, P);
add(“West, new Label(“A”));
add(“South”, new Label(“B”));
```

Questa è la finestra che si ottiene:



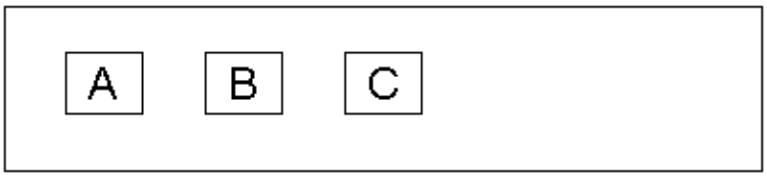
Flow Layout

Il Flow Layout mette i componenti in fila:

- i componenti vengono “ridotti” alla preferredSize

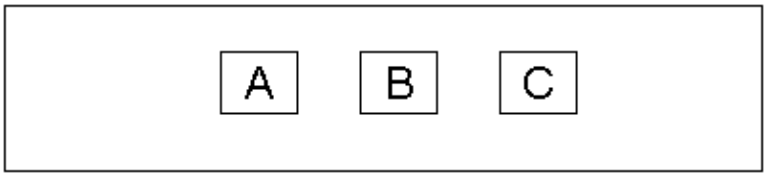
La preferredSize pone la dimensione più adatta al componente e la rende nota.

```
setLayout (new FlowLayout());
add(new Label(“A”));
add(new Label(“B”));
add(new Label(“C”));
```



E’ possibile cambiare allineamento

```
setLayout (new FlowLayout(FlowLayout.CENTER));
add (new Label(“A”));
add (new Label(“B”));
add (new Label(“C”));
```

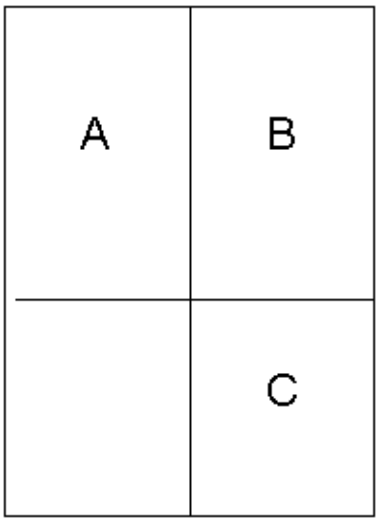


Grid Layout

Il GridLayout dispone i componenti a griglia usando le seguenti specifiche

- il costruttore specifica le dimensioni dei componenti
- per saltare una cella si inserisce un componente vuoto

```
setLayout(new GridLayout(2,2));
add(new Label (“A”));
add(new Label (“B”));
add(new Panel());
add(new Label (“C”));
```



Label e Button

- Label

  Dichiarazione: Label([String label, int align])
  Gli allineamenti ammessi sono: CENTER LEFT RIGHT
  La label corrisponde a una String property text

- Button

Dichiarazione: Button([String label])
La label corrisponde a una String property label
Inoltre la String property risponde all'actionCommand

TextComponent & c.

Vediamo ora la TextComponet e i suoi componenti.

- TextComponent

String property text e selectedText
```
void select(start, end), selectAll()
boolean property editable
int property caretPosition
```

- TextField extends TextComponent

```
TextField extends TextComponent
TextField([String init] [,int size])
char property echoChar
int property columns
```

- TextArea extends TextComponent

```
TextArea extends TextComponent
TextArea([String init, int rows, int cols [, int scrollbar]])
 SCROLLBARS_BOTH, NONE, HORIZONTAL_ONLY, VERTICAL_ONLY
append(String), insert(int pos, String)
replaceRange(String, int, int)
```

Choice e List

- Choice
```
add(String item) insert(String item, int index) remove(int index) removeAll()
void select(int n) int getSelectedIndex() String getSelectedItem()
```

- List
```
add (String item [,int pos]) remove(int index) removeAll() replaceItem()
void select(int n) int getSelectedIndex() String getSelectedItem() setMultipleMode(boole
getSelectedIndexes() String[] getSelectedItems()
```

Checkbox

La Checkbox ha uno stato (on/off) e una label opzionale

```
int property state
String property label
```

La label può appartenere ad un CheckboxGroup assumendo comportamento da RadioButton

CheckboxGroup

```
int property selectedCheckbox
```

Window

E' una classe base comune a Frame e Dialog (si usano in pratica le altre due)
`show() hide()`
occorre chiamare `dispose()` quando una la finestra non è più necessaria
`pack()` causa la ridisposizione dei componenti
`toFront() toBack()`posiziona i componenti

Frame e Dialog

Le Frame e le Dialog estendono *`java.awt.Window`*

- Frame:

  Può avere una `MenuBar ( setMenuBar(MenuBar) )` e una icona `(setIcon(Image))`
- Dialog:

E' una finestra "modale" che blocca l'input per le altre finestre
Il costruttore richiede un parent Frame

Menu

Costruzione di una MenuBar
Innanzitutto si crea una MenuBar
Alla MenuBar si aggiungono dei Menu
Ai Menu si aggiungono dei MenuItem

- checkbox: CheckboxMenuIte
- separatore: Menu.addSeparator()

infine si assegna ad un Frame con setMenuBar()

Thread

- Introduciamo la programmazione parallela, analizzando processi e thread.
- Vediamo innanzitutto le differenze tra il tradizionale multiprocessing e il più moderno multithreading.
- Esamineremo poi come come si fa a creare dei thread
- Tratteremo infine i problemi della sincronizzazione

Multiprocessing

I Multiprocessing sono caratterizzati dai seguenti punti:

- più processi in esecuzione girano in spazi di memoria separati
- comunicazione attraverso primitive (pipe, socket)
- inefficienza dell'interazione (praticamente sono processi in rete)
- maggiore protezione

Multithreading

I Multithreading sono caratterizzati dai seguenti punti:

- più thread in esecuzione girano nello stesso spazio di memoria
- comunicazione attraverso memoria condivisa
- comunicazione molto efficiente e potente
- problemi di sincronizzazione

Creazione

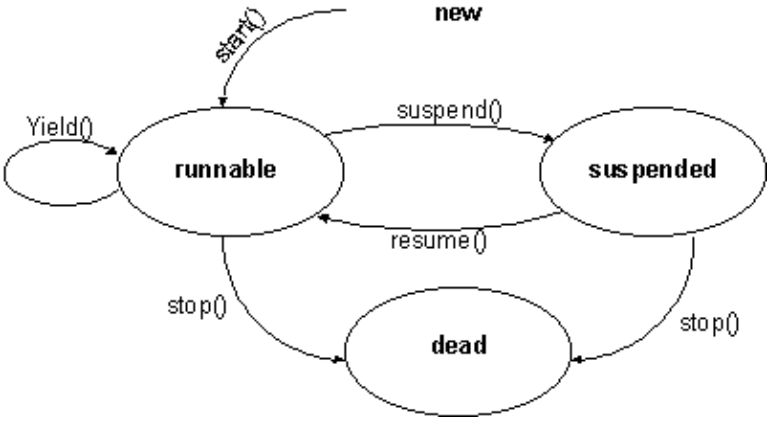
Vediamo ora come si crea un Thread:

1. Inanzitutto bisogna estendere la classe Thread
2. Ridefinire il metodo run()
3. Avviare il thread con start()

```
class Contatore1 extends Thread {
    public void run() {
        int n = 0;
        while(true) {
            System.out.println(""+n);
            ++n;
        }
    }
    public static void main(String[] args) {
       new Contatore1().start();
    }
}
```

Stati

Graficamente possiamo dire che gli stati dei Thread sono:



Vediamo in dettaglio gli stati dei Thread:

1. Un thread appena creato è nello stato new non è attivo: non fa nulla
2. Per attivare un thread, occorre chiamare start() il thread è runnable
3. un thread runnable ottiene ogni tanto il processore (o uno dei processori)
4. un thread runnable può cedere il passo agli altri con yield() rimanendo attivo
5. Un thread attivo può andare in stato di suspended (non ottiene mai il processore) con suspend()
6. Un thread sospeso ritorna in esecuzione con resume()
7. Un thread può morire (e non più ritornare in esecuzione) con stop()

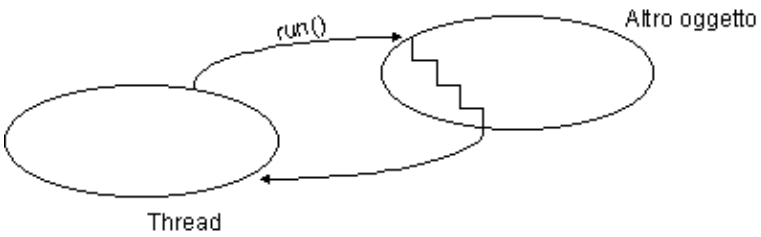
Runnable

Spesso si vuol rendere autonomo un oggetto già derivato da qualcos'altro.
Per esempio immaginiamo una finestra: è un oggetto GUI, deve derivare da Component

Interfaccia Runnable:

```
import java.awt.*;
class Contatore2 extends Frame
   implements Runnable {
      Contatore2() {
         …  // inizializza la GUI
        new Thread(this).start();
      }
      public void run() {
         int n = 0;
         while(true) {
           label.setText(""+n);
           ++n;
         }
      }
      public static void main(String[] args) {
          Contatore2().start();
      }
}
```

Si vede quindi come si rende multithreaded un oggetto non estendibile creando un nuovo oggetto thread
Il thread si "aggancia" all'oggetto originario eseguendo il metodo run



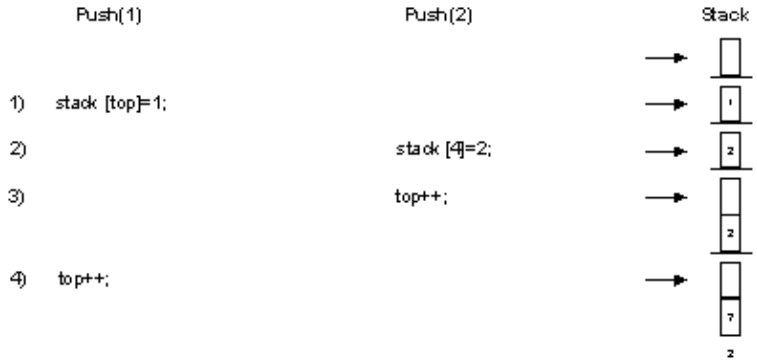
Questo esempio mostra che le interfacce sono un sostituto per i puntatori a funzione.

Problema!

La programmazione concorrente riserva brutte sorprese. Consideriamo uno stack:

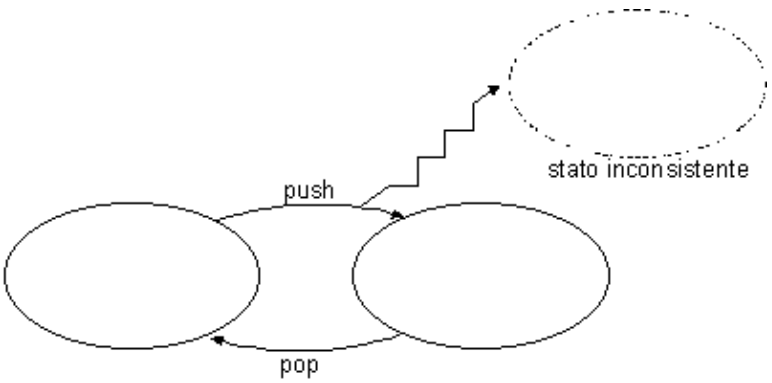
```
class Stack {
   int top; int stack = new int[10];
   void push(int x) { stack[top]=x; top++; }
   int pop() { return stack[--top];
}
```

Supponiamo una sequenza di esecuzione sfortunata di due push sullo stesso stack da parte di due thread:



Perchè?

Il problema di prima è tipico della programmazione multithreaded
L'operazione di push è atomica:



In generale i metodi portano un oggetto da uno stato consistente ad un altro stato consistente.
Il multithreading consente ad un thread di accedere ad un oggetto mentre è in uno stato inconsistente.

Sincronizzazione

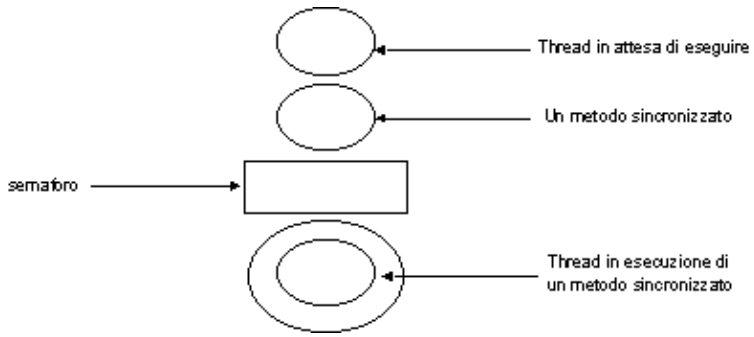
- Si deve garantire che i metodi che eseguono metodi critici non siano interrotti
- Il problema dello Stack in Java ha una soluzione semplice:

```
class Stack {
   synchronized void push(int x) { … }
   synchronized  int pop() { … }

}
```

- Questo garantisce che all'oggetto acceda un thread solo per volta
- Cosa fa in realtà synchronized è una cosa più complessa…

Sincronizzazione



- Un oggetto con metodi sincronizzati ha associato un semaforo.
- Quando un thread accede ad un oggetto sincronizzato, sbarra l'accesso a tutti gli altri
- Ogni nuovo thread viene posto in attesa (sospeso)
- Quando un thread finisce l'accesso, viene riattivato il primo in attesa
- Attenzione: la sincronizzazione ha un impatto *notevole* sull'efficienza.

Note

- La sincronizzazione di un metodo normale è legata ad una specifica instanza di un oggetto, non a una classe.
- La sincronizzazione di un metodo statico blocca l'accesso a tutti gli altri metodi statici (non dinamici).
- Se accedo ad un oggetto chiamando un metodo (per esempio push) sbarro l'accesso all'oggetto utilizzando qualunque altro metodo (per esempio anche pop)
- Se si ridefinisce un metodo sincronizzato, il metodo ridefinito non è automaticamente sincronizzato:
  - ♦ può esserlo, ma va dichiarato esplicitamente
  - ♦ può non esserlo; allora chiamando il padre con super si entra in un metodo sincronizzato

Wait e notify

Problema: e se un thread mentre accede a un metodo sincronizzato si accorge di non poter finire il lavoro?

Esempio:

```
class Stack {
   synchronized void push(int x) {
      if(top>stack.length)
        // che faccio ???
   }
}
```

- Posso uscire senza fare niente oppure sollevare una eccezione
- La cosa migliore sarebbe aspettare che qualcuno faccia una pop
- La `wait()` e la `notify()` fanno parte del linguaggio: sono metodi di Object.
  - `wait()` ha senso solo in un metodo sincronizzato:
    Sospende il thread corrente e lo pone in attesa
  - Un altro thread va in esecuzione:
    il thread viene riattivato non appena qualcuno effettua una `notify()` sull'oggetto.
    Un thread riattivato accede all'oggetto dal punto in cui si trovava non appena ottiene il semaforo

```
class Stack {
   synchronized void push(int x) {
      while(top>stack.length)
        wait();
        …
   }
   synchronized void pop(int x) {
       int r = stack[--top];
       notify();
       return r;
   }
}
```

Note

- Il ciclo while non può diventare un if : non è detto che dopo una `notify()` le condizioni siano soddisfatte
- La notify riattiva un solo thread
- Per riattivare tutti i thread in attesa si deve usare `notifyAll()`
  Però e' più inefficiente.

```
class Stack {
   int[] stack = new int[10]; int top=0;
   synchronized void push(int x) {
      while(top>stack.length)
        wait();
      stack[top++]=x;
      notify();
   }

   synchronized void pop() {
       while(top==0)
 wait();
       int r = stack[--top];
       notify();
       return r;
   }

}
```

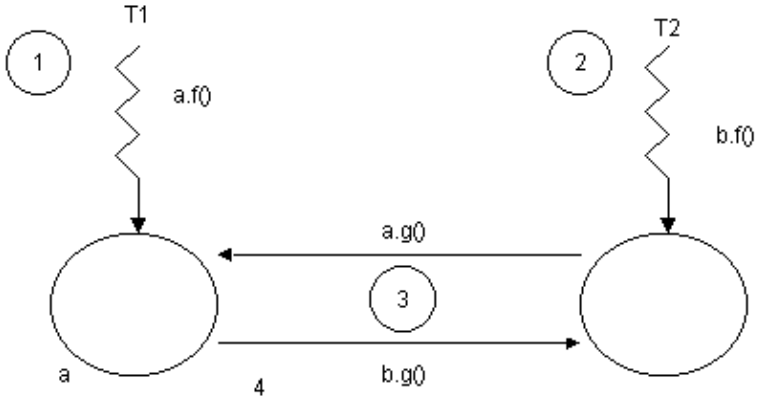
Esempio

```
class Stack {
   int[] stack = new int[10];
   int top=0;
   synchronized void push(int x) {
      while(top>stack.length)
        wait();
      stack[top++]=x;
      notify();
   }
}
```

```
class Stack {
   synchronized void popOdd() {
    while(top==0 || stack[top]%2==1)
 { notify(); wait(); }
    int r = stack[--top];
    notify();
    return r;
   }
   synchronized void popEven() {
    while(top==0 || stack[top]%2==0)
 { notify(); wait(); }
    int r = stack[--top];
    notify();
    return r;
   }
}
```

Stallo

Lo stallo è il principale problema della programmazione concorrente.



T1 esegue a.f() e blocca a
T2 esegue b.f() e blocca b
T1 chiama b.g() ma b è bloccato, e si sospende
T2 chiama a.g() ma a è bloccato, e si sospende
STALLO: T1 e T2 sono sospesi ognuno in attesa che l'altro faccia qualcosa.

Socket

Un Socket è un oggetto che consente di creare Connessione di rete

Vediamo come si costruisce un socket client:

```
Socket(String host, int port) Socket(InetAddress add, int port)
```

In questo modo si estraggono gli stream:

```
InputStream getInputStream() InputStream getOutputStream()
```

Socket

Diamo alcune informazioni generali

```
InetAddress getInetAddress()
int getPort() int getLocalPort()
```

Timeout

```
int getSoTimeout()
void setSoTimeout(int)
```

ServerSocket

Un server socket accetta connessioni: `ServerSocket(int port)`
Accetta connessioni: `Socket accept() void close()`
Timeout: `int getSoTimeout() void setSoTimeout(int)`

InetAddress

InetAddress rappresenta un indirizzo internet:
`static InetAddress.getByName (String host)` con parametro IP o nome DNS
Informazioni:
`String getHostName()`
`String getHostAddress()`

URL

I costruttori sono:
`URL(String url)`
`URL(String protocol, String host,[int port], String file)`
Per scandire le parti di un URL abbiamo:
`String getProtocol(), getFile()`…
Per le connessioni:
`URLConnection openConnection()`
`InputStream openStream()`
`Object getContent()`

URLConnection

L'URLConnection viene costruito tramite
```
URL.openConnection()
InputStream getInputStream()
OutputStream getOutputStream()
Object getContent()
String getContentEncoding()
int getContentLength()
long getExpiration()
long getDate()
```

La URLConnection ritorna gli header della connessione:
```
String getHeaderField   (String name| int n)
Date getHeaderFieldDate   (String name, long default)
int getHeaderFieldInt   (String name, int default)
String getHeaderFieldKey(int n)
```

URLEncoder

La `static URLEncoder.encode(String s)` codifica una stringa nel formato URL (formato canonico in subset portabile ASCII): spazi in + caratteri non alfanumerici in %<hex>

DatagramSocket

DatagramSocket manda messaggi UDP asincroni ai socket
```
DatagramSocket([int port])
void send(DatagramPacket p)
void receive(DatagramPacket p)
void close()
```

DatagramPacket

DatagramSocket manda messaggi UDP asincroni ai packet
```
DatagramPacket(byte[] buf, int len[, InetAddress addr, int port])
setAddress(InetAddress addr)
byte[] getData()
void setData(byte[] data)
```