Package

- Un package è una collezione di classi logicamente connesse che possono interagire tra loro.
  - Meccanismo per raggruppare classi in **librerie**.
- Hanno struttura gerarchica per evitare collisioni di nomi (diversi programmatori possono dare lo stesso nome a classi differenti).
- Le classi di un package possono:
  - essere incapsulate anche a livello di file
  - condividere dati e metodi con classi di altri package
  - implementare oggetti distribuiti

Package

- Alcuni tra i principali package di Java:
  - **`lang`**: funzioni comuni del linguaggio
  - **`util`**: altre funzioni di servizio
  - **`io`**: input/output
  - **`text`**: formattazione testi
  - **`awt`**, **`javax.swing`**: grafica, interfacce grafiche
  - **`awt.event`**: gestione eventi
  - **`applet`**: programmi da eseguire in un browser
  - **`net`**: gestione di rete

Package

- Con i package si possono raggruppare più classi correlate tra loro, ad esempio le classi di una stessa applicazione.
- Un file che contiene codice sorgente Java si chiama **unità di compilazione.**
  - Di solito una classe = un file, ma non è necessario
  - Il vincolo obbligatorio è che ogni file abbia al più una classe di tipo **`public`**; questa deve avere lo stesso nome del file.

Package

- Per dichiarare che una classe appartiene a un package bisogna usare l'istruzione:

```
package nome_package;
```

- L'istruzione deve essere necessariamente la prima che compare nell'unità di compilazione.
  - Possono precederla solo linee vuote o commenti.
- Se ci sono più classi nella stessa unità di compilazione queste vengono tutte incluse nello stesso package.

Package

- I package archiviano le classi rispecchiando la struttura del file system.
- Si usa il punto `.` come separatore.
  - Il package **`ling2.prog.esercizi`** avrà i bytecode delle classi in **`ling2/prog/esercizi`**
- I package che iniziano per **`java`** contengono classi che sono parte del linguaggio.
- Quindi le *nuove classi* devono essere contenute in package con nomi differenti.

Package

- Per potere utilizzare le classi di un package, bisogna che **`CLASSPATH`** ne contenga il path.
- La variabile **`CLASSPATH`** deve contenere la root del path, non direttamente le classi.
  - Ad esempio per usare **`ling2.prova`**, le cui classi sono in **`C:\java\lib\ling2\prova\*`** bisogna che **`CLASSPATH`** contenga **`C:\java\lib`**
- In MS Windows, **`CLASSPATH`** è di questo tipo:

```
CLASSPATH = C:\Program Files\Java\lib;
C:\Program Files\Java\import;.;.\;
```

Package

- Ad esempio, definiamo una classe A come:

```
package pack1;

public class A
{ public String toString()
   { return "class pack1.A";
   }
}
```

- Il *nome completo* della classe è dato da **`nome_package.nome_classe`**
  - Nell'esempio è `pack1.A`

Importazione

- Per utilizzare la classe si può usare
  - il suo nome completo (qualificato), **`pack1.A`**
  - il solo nome della classe se si è invocata la direttiva di importazione (**`import pack1.A`**)
- L'importazione non fa altro che sostituire il nome qualificato *durante la compilazione*
- Ad esempio **`import pack1.A`** dice al compilatore di aggiungere **`pack1.A`** allo spazio dei nomi usato, per identificare **`A`**.

Importazione

- L'importazione consente di semplificare il riferimento alla classe, altrimenti occorrerebbe sempre specificare il nome del package
  - Cosa scomoda soprattutto se la gerarchia del package contiene più livelli
- Le classi andrebbero importate una per una; in alternativa, si può anche usare l'*importazione su richiesta*:

```
import ling2.prog.esercizi.*
```

Importazione

- La sintassi è fissa: non consente altre forme.
  - L'asterisco non va letto come "simbolo jolly".
- Per esempio non si possono importare tutte e solo le classi che cominciano per S scrivendo:

```
import ling2.prog.esercizi.S*; //NO!
```

- Il compilatore inoltre importa automaticamente:
  - le classi del package **`java.lang`**
  - le classi che si trovano nella directory in cui si esegue la compilazione

Importazione

- L'importazione può creare **conflitti di nomi**.
- Se esiste una classe di nome `A` dentro ai package `pack1` e `pack2`, che cosa capita con:

```
import pack1.*;
import pack2.*;
```

POSSIBILE CONFLITTO

- Se nel codice successivo si scrive solo `A`, il compilatore lamenterà che non sa identificare quale classe è identificata dal nome `A`.

Importazione

- Modi che non portano a un conflitto:
  - usare i nomi qualificati per le due classi (cioè scrivere sempre `pack1.A` oppure `pack2.A`)
  - importare solo un package e usare il nome completo per la classe dell'altro package
- In generale, quando si importano package è bene controllare eventuali possibili conflitti.
- Quando si scrivono classi è bene sincerarsi che i loro nomi non siano in conflitto con quelli di classi note di altri package.

Visibilità

- L'uso dei package serve anche a proteggere l'ambiente di sviluppo del codice (ovvero: nascondere dettagli all'esterno)
- Ciò è realizzato con i modificatori di visibilità.
- Di default, una classe può essere utilizzata solo da altre classi **del suo stesso package**.
- Tuttavia è possibile dire che alcune classi possono essere visibili all'esterno: **`public`**.

Visibilità

- Si può dichiarare **`public`** al più una classe per unità di compilazione (=file sorgente)
  - la classe pubblica è l'unica le cui istanze e metodi statici possono essere invocati fuori dal package
  - le altre classi sono dettagli di implementazione
- Se ad esempio mettiamo in un solo file le classi **`A`** (classe pubblica), **`B`**, **`C`**:
  - il file si deve chiamare **`A.java`**
  - quando **`A.java`** viene compilato, si producono **tre** diversi file: **`A.class`**, **`B.class`**, **`C.class`**

Visibilità

- Pubblicare la classe non rende pubblici i suoi membri: anche la classe pubblica può comunque avere membri privati
  - Se si definisce **`private`** un membro della classe se ne proibisce l'accesso a **ogni** altra classe
  - Se non si mette niente, resta comunque solo l'accessibilità a livello di package
  - Solo usando il modificatore **`public`** per il membro (oltre che per la classe) questo è accessibile da chiunque (sia fuori che dentro il package)

Esempio

```
package sample;
public class Visible
{ public int x;
  int y;
  public Visible() { this(1,2); }
  Visible(int a, int b) { x = a; y = b; }
  public String visibileFuori()
  { return "metodo visibile fuori"; }
  String nascostoFuori()
  { return "metodo nascosto fuori"; }
}
```

Esempio

- In questo package la classe **`Visible`** è dichiarata **`public`**
  - il campo **`x`**, il costruttore senza argomenti (che invoca quello con due), il metodo **`visibileFuori`** sono **`public`**
  - il campo **`y`**, il costruttore con due argomenti, il metodo **`nascostoFuori`** sono visibili all'interno del package, ma non all'esterno

Esempio

- Quindi se scriviamo:

```
import sample.*;
class Prova
{ public static void main(String[] args)
  { Visible v = new Visible(3,3);
    System.out.println("y = " + v.y);
    System.out.println(v.nascostoFuori());
}
```

ci saranno degli errori?

Esempio

- Ci sono errori: il compilatore dirà che:
  - non si può usare il costruttore con due argomenti fuori dal package.
  - non si può accedere il campo **`y`** fuori dal package
  - non si può invocare il metodo **`nascostoFuori`** fuori dal package
- Discorso analogo se definiamo una classe del package non come classe pubblica di una unità di compilazione, denotata da "**`public class`**", ma come semplice "**`class`**"

Visibilità

- Se scriviamo:

```
package sample;
public class Visible ...
class Invisible ...
```

le due classi differiscono per visibilità all'esterno del package (**`Visible`** è visibile, **`Invisible`** no).

Visibilità

- Anche se definissimo un metodo di **`Invisible`** come **`public`**, questo non sarebbe accessibile all'esterno del package.
- Quando compiliamo classi non esplicitamente incluse in un package, Java le tratta come se appartenessero ad un *package di default*, **senza nome**, che include tutte le classi e tutte le interfacce definite nella stessa directory.

Array

- Molte applicazioni devono memorizzare molteplicità di dati omogenei, ossia **collezioni** di dati (in particolare, liste, code...); sono richieste strutture dati appropriate.
- Java mette a disposizione:
  - **array**, implementati direttamente a livello di linguaggio
  - altre strutture avanzate (come le collezioni) disponibili nelle librerie e con funzioni avanzate già implementate

Array

- Un array è una sequenza finita di variabili dello stesso tipo. Questo tipo è detto **tipo base**.
  - Definire un array come sequenza vale a dire che gli elementi sono disposti in modo ordinato
  - Ogni elemento è accessibile specificando la posizione in cui si trova (**indice**)
  - La finitezza corrisponde a richiedere che esista un dato intervallo per gli indici.

Array

- In Java, gl'indici devono essere numeri interi e la loro numerazione **parte da zero**.
  - Motivi storici oramai obsoleti per cui un array di `n` elementi contiene elementi con indici da `0` a `n-1`
- In Java si richiede che le dimensioni (cioè il valore di `n`) vadano specificate a priori
- In Java il tipo base degli array può essere qualunque, sia primitivo sia riferimento.
  - Sono possibili "array di oggetti"

Array e collezioni

- Si possono superare queste limitazioni con strutture dati più potenti (es. collezioni)
  - Rendono possibile allocazione dinamica delle dimensioni finite
  - Hanno già implementate diverse funzioni
- Strutture non native del linguaggio, ma disponibili in librerie standard: introduzione trasparente

Array e collezioni

- Prezzi da pagare:
  - notazione meno semplice
  - non si possono usare per i tipi primitivi, solo gli oggetti (tipi riferimento)
- La soluzione al secondo problema è quella di usare le **classi involucro**
  - dato che le collezioni di **`int`** non sono ammesse, vanno implementate come collezioni di oggetti di tipo **`Integer`**

Array

- In Java la presenza di array è rivelata dalle parentesi quadre `[]` . Esse servono:
- per denotare il tipo riferimento ad un array
  - **`int[]`** = riferimento array di tipo base **`int`**
  - **`String[]`** = riferimento array di tipo base **`String`**
- come operatore di indicizzazione dell'array
  - se ho un array di **`String`** chiamato **`nomi`** i suoi elementi sono **`nomi[0], nomi[1], ...`**

Gestione array

- In Java la gestione degli array è simile a quella degli oggetti istanza di una classe.
- Gli array vanno:
  - dichiarati in una variabile (di **tipo riferimento**); la dichiarazione non crea di per sé l'oggetto, ma un riferimento a **`null`**
  - creati in memoria; a questo punto la variabile riferisce un'area di memoria (ancora vuota)
- Una volta creati, gli array si possono accedere e modificare tramite l'operatore **`[]`**

Gestione array

- Così come per gli oggetti, le variabili di tipo riferimento degli array vanno riempite con **`new`**
- La sintassi però è diversa dagli oggetti:

```
new tipo_base[numero_elementi]
```

- dove:
  - `tipo_base` denota quale genere di elementi sarà memorizzato nell'array
  - **`numero_elementi`** dev'essere un'espressione riconducibile a un valore intero

Gestione array

- Tipicamente si scrive quindi:

```
String[] nomi = new String[4];
int[] numbers = new int[10];
```

dichiarazione — creazione

  - ammesso anche "`int numbers[] = new ..`" ma è una sintassi sconsigliata (meno leggibile)
- La creazione dell'array non crea 4 stringhe (o 10 int), ma una dimensione in memoria dove si **possono memorizzare** 4 stringhe (o 10 int).

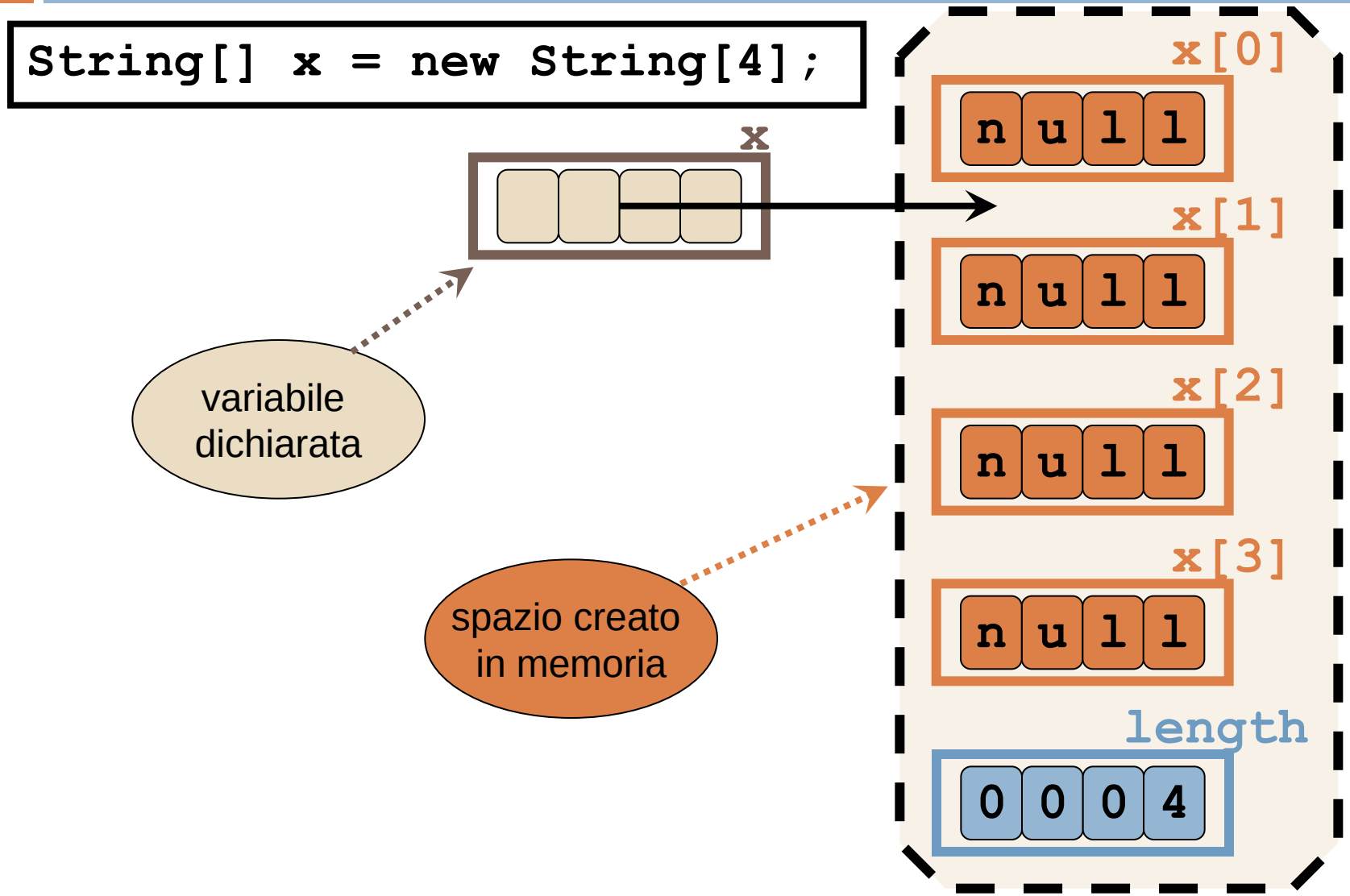
Campo `length`

- Subito dopo la creazione, l'area di memoria dell'array è come un oggetto. Contiene:
  - i campi dell'array (vuoti)
  - un campo aggiuntivo di nome **`length`** e tipo **`int`**.
- **`length`** = # di elementi: viene fissato alla creazione e non può essere cambiato.
  - vantaggio: evitare una variabile aggiuntiva

```
for(i=0; i<x.length; i++) { x[i] = "A" + i; }
```

  - non va però confuso con il metodo **`length()`** !!

Gestione array



Operatore di selezione

- L'operatore di selezione `[]` è un op. binario:
  - primo operando: riferimento all'array
  - secondo operando: espressione intera tra `[]`
- Quest'ultima è detta **indice** o **selettore**.
  - Dev'essere un **`int`**. Vanno bene anche **`byte`**, **`char`**, **`short`**(viene fatto un cast) ma non **`long`**.
  - Dev'essere compresa tra **`0`** e **`length-1`** !
  - È molto frequente sbagliarsi e usare **`length`**, che non è un valore valido, o finire "in negativo".

Array fuori dai limiti

- Cosa succede in questo caso?
- Si verifica un errore **al tempo di esecuzione** di tipo **`ArrayIndexOutOfBoundsException`**.
- Avviene a run-time perché non può essere controllato dal compilatore (che non conosce i valori che vengono passati all'operatore `[]`)
- Solo la JVM è in grado di accorgersene.

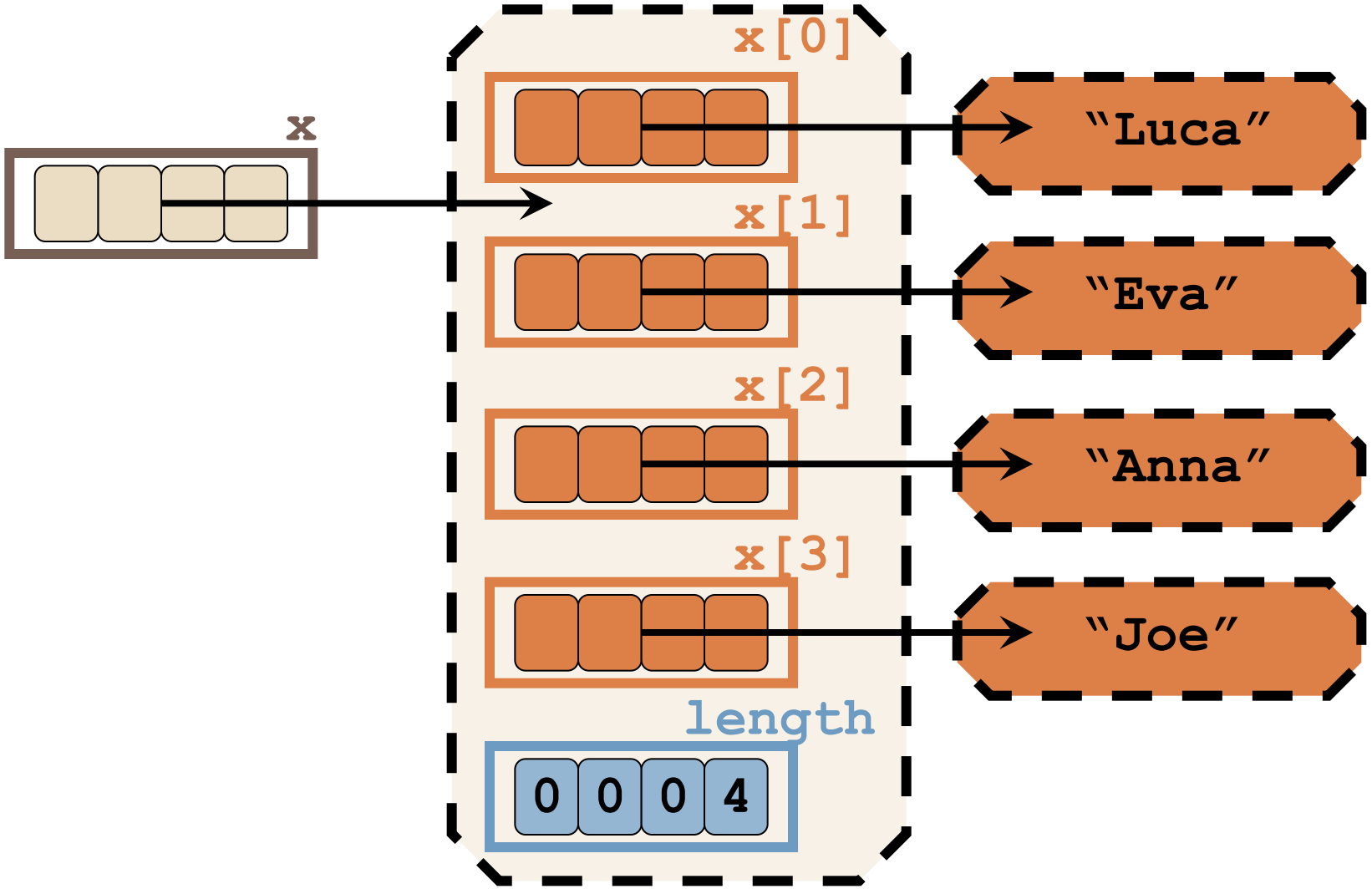
Inizializzazione

- È possibile inizializzare direttamente l'array insieme alla dichiarazione, specificandone i valori tra parentesi graffe: **{}**.

```
String[] x = {"Luca","Eva","Anna","Joe"};
```

- Compatta creazione della variabile riferimento array e creazione dell'area di memoria.
- La dimensione dell'array (**x.length**) viene dedotta dal compilatore contando gli elementi.

Inizializzazione



Uso dei valori

- Le espressioni dove si recupera un elemento dall'array si possono usare ovunque ha senso usare una variabile del tipo base dell'array.

```
String[] x = new String[4];
int[] n = new int[10];
...
x[2] = "Andrea";
double triplohead = n[0];
if (x[4].equals("Giulia") ...
int a = n[n[n.length/2]];
```

potrebbe dare errore in esecuzione, ma per il compilatore è corretta... cosa fa esattamente?

Uso dei valori

- Usare un campo di un array a sinistra di un assegnamento è un metodo **set** per array.
  - Il contenuto dell'array viene modificato.

```
numbers[5] = 4;
```

- Usare un campo di un array a destra di un assegnamento è un metodo **get** per array.
  - Il contenuto viene acceduto senza modifiche.

```
stringa = nomi[2];
```

Array come parametri

- I parametri formali di metodi e costruttori possono essere di tipo array.
- L'esecuzione di un metodo a cui viene passato un array come parametro può modificarne il contenuto come effetto collaterale.
  - Questo perché viene passato per valore il contenuto della variabile riferimento array.
  - Quindi se ne possono accedere i campi e cambiarne il valore.

Array come parametri

- Anche per gli array può essere utile usare un costrutto di array anonimo, senza cioè memorizzarne il contenuto in una variabile.

```
s = somma(new int[] {4,8,15,16,23,42});
```

- Pure il tipo di ritorno di un metodo può essere di tipo array.

```
public String[] listaNomi(int[] posiz)
```

Il metodo `main`

- Il metodo **`main()`** ha sempre un solo parametro (**`args`**), di tipo array di **`String`**.
  - Questo contiene eventuali parametri passati all'esecuzione come **`args[0]`**, **`args[1]`**, ...
  - Quanti sono? **`args.length`** !
  - Il valore di **`args`** viene stabilito dalla JVM quando viene lanciata l'esecuzione.

Confronto tra array

- Per confrontare array, non va bene ==.
  - Sono tipo riferimento! Serve un metodo **`equals`**
  - Questa e altre funzionalità sono fornite dalla classe **`Arrays`**.
  - Ad esempio, c'è un metodo statico di tipo equals: **`Arrays.equals(a,b)`** che ritorna **`boolean`**
  - Per poterlo eseguire, i due array devono essere dello stesso tipo.

Array multidimensionali

- Si possono creare array multidimensionali

```
int[][] x = new int[3][10];
```

- Array bi-dimensionale (matrice): esistono anche a tre, quattro,... dimensioni ma sono molto più rari.
- In realtà una matrice non è altro che un array di array. Nell'esempio di sopra, ci sono 3 righe (`x[0]`, `x[1]`, `x[2]`) di tipo `int[]` (ognuna con 10 elementi).

Costrutto for-each

```
Studente[] x = new Studente[4];
x[0].nome = "Luca"; x[1].nome = "Ugo"; ...
```

- Tipicamente gli elementi di un array si accedono con cicli-for.
- Java consente una forma abbreviata detta for-each in cui non occorre usare un indice i.

```
for (int i=0; i<x.length;i++)
{ System.out.println(x[i].nome); }
```

```
for (Studente s : x)
{ System.out.println(s.nome); }
```