Esercitazioni del corso di:

# ALGORITMI E STRUTTURE DATI

Tutor: Francesca Piersigilli

email: francesca.piersigilli@unicam.it

# Strutture dati elementari

Tecniche di organizzazione dei dati: scelta della struttura dati

Scelta strutture dati + scelta algoritmo

=

Guadagno in tempo e spazio

- Non sono oggetti passivi
- Considerare sempre le operazioni necessarie per manipolare le strutture dati (algoritmi)

# Costrutti di base I

•*Tipi* di dati: ci consentono di specificare come sono organizzati insiemi di bit che costituiranno il dato

•*Metodi*: definiscono operazioni su quei dati

•*Classi*: descrivono le informazioni che elaboriamo, i metodi che agiscono sulle informazioni, costruiscono gli oggetti

Tutte le strutture dati sono oggetti e possono essere riferimenti ad altri oggetti

# Costrutti di base II

Valori booleani (**boolean**)
Caratteri (**char**)
Numeri interi a 8 bit (**byte**)
Numeri interi a 16 bit (**short**)
Numeri interi a 32 bit (**int**)
Numeri interi a 64 bit (**long**)
Numeri in virgola mobile a 32 bit (**float**)
Numeri in virgola mobile a 64 bit (**double**)

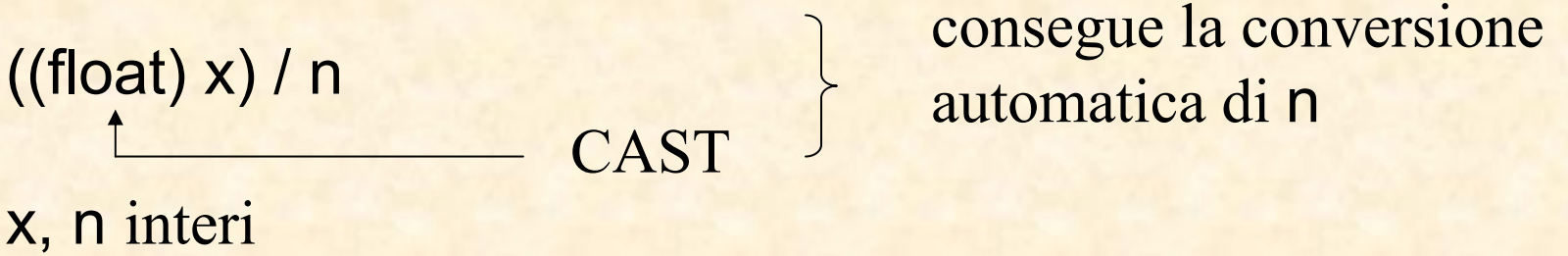
Tra parentesi i rispettivi nomi in Java

# Costrutti di base III

In seguito utilizzeremo tipi **int** e **double**

**Definizione**: un *tipo di dato* è definito da un insieme di valori e da una collezione di operazioni su questi valori.

Quando si esegue un'operazione è bene essere sicuri che sia gli operandi, sia il valore di ritorno siano del tipo corretto. In alcuni casi, quando così non è, Java esegue conversioni di tipo in modo automatico, oppure è il programmatore a utilizzare esplicitamente l'operatore *cast*.



# Costrutti di base IV

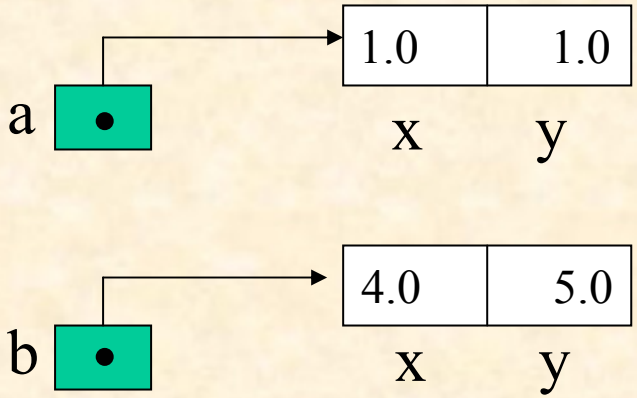
Spesso abbiamo bisogno di definire i *nostri* tipi di dati per organizzare i programmi in modo efficace.

In un certo senso ogni programma Java è un tipo di dato, poiché astrattamente possiamo vederlo come una lista di valori (*attributi primitivi o meno*) con operazioni associate (*metodi*).

Esempio:

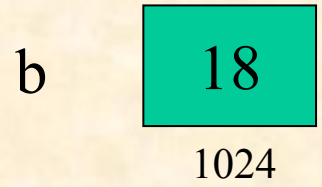
```
class Point {
        double x, y;
}

Point a = new Point(), b = new Point();
a.x = 1.0; a.y = 1.0;
b.x = 4.0; b.y = 5.0;
```

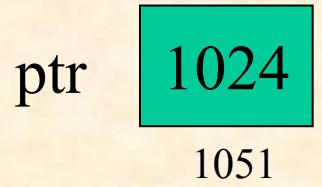


# Riferimenti

Quando creiamo una variabile di tipo *int* o *double* oppure di qualsiasi altro tipo primitivo, il nome che adottiamo è un sinonimo di una locazione di memoria contenente informazioni su quella variabile;




Per tutti gli altri oggetti, il nome è sinonimo di quello che in Java viene chiamato *riferimento*. Un riferimento specifica l'indirizzo di una locazione di memoria che contiene le informazioni.



# Riferimenti (passaggio argomenti)

Il riferimento è un concetto molto importante, in quanto permette di strutturare i dati, in modo da supportare per essi algoritmi di elaborazione efficienti.

I riferimenti sono in effetti, la base costitutiva di molte strutture dati e di molti algoritmi.

Ad esempio il riferimento rende agevole il passaggio di oggetti come parametro ai metodi:

```
double distance (Point a, Point b){
        double dx = a.x – b.x;
        double dy = a.y – b.y;
        return Math.sqrt(dx*dx + dy*dy);
}
```

# Riferimenti - puntatori

Nel linguaggio C e in molti altri linguaggi, i riferimenti sono noti con il nome di *puntatori*, e vengono spesso utilizzati.
In Java, invece, non c'è distinzione fra un riferimento ed un oggetto, quindi non c'è altra scelta che accedere agli oggetti tramite riferimenti.

# Esercizio

Definire una classe che sia adatta a rappresentare il gioco delle carte, che includa metodi per controllare se due carte sono dello stesso seme e per verificare se una ha valore maggiore dell'altra.

# Soluzione

```
class Cartadagioco{
        int valore;
        String seme;
        public Cartadagioco maggiore(Cartadagioco a, Cartadagioco b){
                return (a.valore < b.valore ? b : a);
        }

        public boolean stessoseme(Cartadagioco a, Cartadagioco b){
                return (a.seme == b.seme ? true : false);
        }
}
```

# Array

Insieme fissato di elementi dello stesso tipo memorizzati in modo contiguo e accessibili per mezzo di un indice.
Si denota l'i-esimo elemento dell'array a con a[i].
Gli indici dovranno essere numeri interi non negativi, minori della dimensione dell'array.

Gli array sono molto importanti in quanto hanno una corrispondenza diretta con la struttura della memoria centrale.

Nel linguaggio macchina, per ottenere il valore di una "casella" di memoria è necessario specificarne l'indirizzo; quindi si potrebbe pensare alla memoria di un computer come ad un array nel quale gli indici corrispondono agli indirizzi delle "caselle" di memoria.

# Array

E' possibile passare un riferimento ad un array come parametro a un metodo, e quindi consentire a quel metodo di accedere agli elementi dell'array senza doverne fare una copia.
Questa possibilità è indispensabile quando abbiamo a che fare con array di grandi dimensioni.

```
public boolean search (int a[], int i, int max){
        int index;
        for(index=0; index<max; index++;)
                if(a[index]==i) return true;
        return false;
}
```

Possibile chiamata al metodo:

```
search (arr, i, max);
```

# Array: new

Durante l'esecuzione new alloca la quantità di memoria di cui il nostro array ha bisogno, restituendo un riferimento a tale array.

*Allocazione dinamica*: strumento essenziale per programmi che devono manipolare array diversi, specialmente se di grandi dimensioni.
Senza di essa bisognerebbe preallocare array più grandi possibili.

# Array: Vector

Corrispondenza con i vettori matematici.
**Classe Vector**: oggetto astratto che può essere indicizzato come un array, ma che può anche ridursi e crescere..
Classe che porta vantaggi: è possibile l'ingrandimento e la riduzione senza conoscere il codice per fare queste operazioni.

A differenza degli array, in cui per indicare l'i-esimo elemento si doveva ricorrere all'uso delle [], con Vector basta invocare il metodo get.

# Esercizio I

Sia a un array. Supponete che a sia dichiarato così:

```
int [] a = new int [99]
```

Fornite il contenuto dell'array a seguito dell'esecuzione delle due istruzioni seguenti:

```
for (i = 0; i < 99; i++) a[i] = 98-i;
for (i = 0; i < 99; i++) a[i] = a[a[i]];
```

# Esercizio II

Scrivete un programma che conti, data una sequenza di numeri in ingresso (array di interi), il numero di interi minori di 1000.

# Liste concatenate I

Utile se necessitiamo di scandire una collezione di elementi in modo sequenziale, uno dopo l'altro.

Ogni elemento contiene informazioni per accedere all'elemento successivo.
**Vantaggio**: rispetto all'array, la lista concatenata è più flessibile a modifiche.
**Svantaggio**: per accedere ad un determinato elemento della lista bisogna scorrerla tutta.

Definizione: una lista concatenata è un insieme di elementi, dove ogni elemento è inserito in un nodo contenente anche un link (connessione o riferimento) a un altro nodo.
Un link potrebbe puntare anche al nodo da cui parte (struttura circolare).

# Liste concatenate II

Il link del nodo finale:
- È un link nullo che non punta ad alcun nodo
- Punta ad un nodo fittizio che non contiene alcun elemento
- Punta indietro al primo nodo della lista, creando quindi una lista circolare

In ognuno di questi casi, seguendo i link dal nodo iniziale a quello finale, visitiamo in sequenza tutti gli elementi.

# Rappresentazione in Java

Rappresentazione in Java: usare oggetti per i nodi e riferimenti a oggetti per i link.

```
class Node{
        Object item;
        Node next;
}
```

Implementazione dei nodi delle liste come oggetti di tipo Node: ogni nodo è composto da un elemento (di tipo Object) e da un riferimento a un altro nodo. Quindi, usiamo riferimenti a nodi per implementare link.