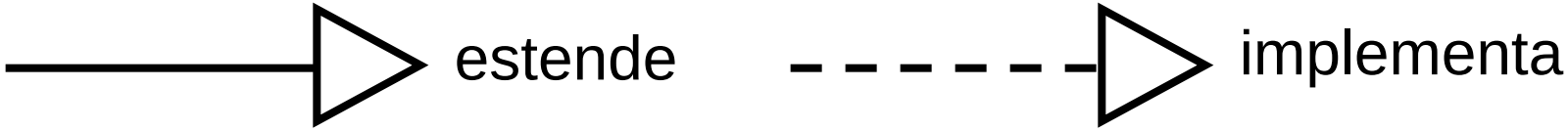
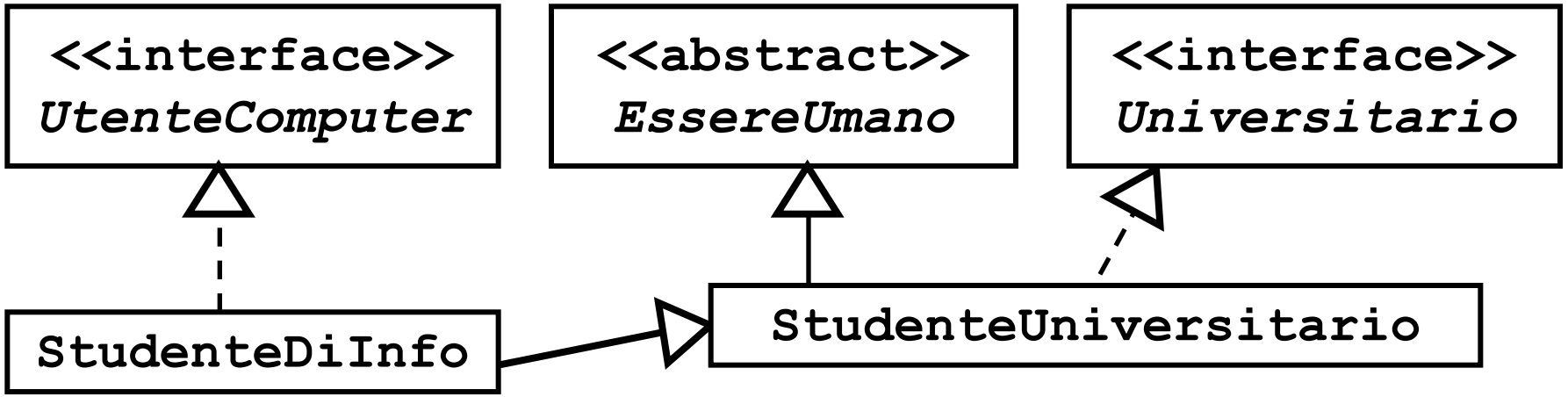
Notazione grafica UML

- Estensione e implementazione in UML:



- Per classi astratte e interfacce i nomi vanno in corsivo preceduti da **<<abstract>>** e **<<interface>>**

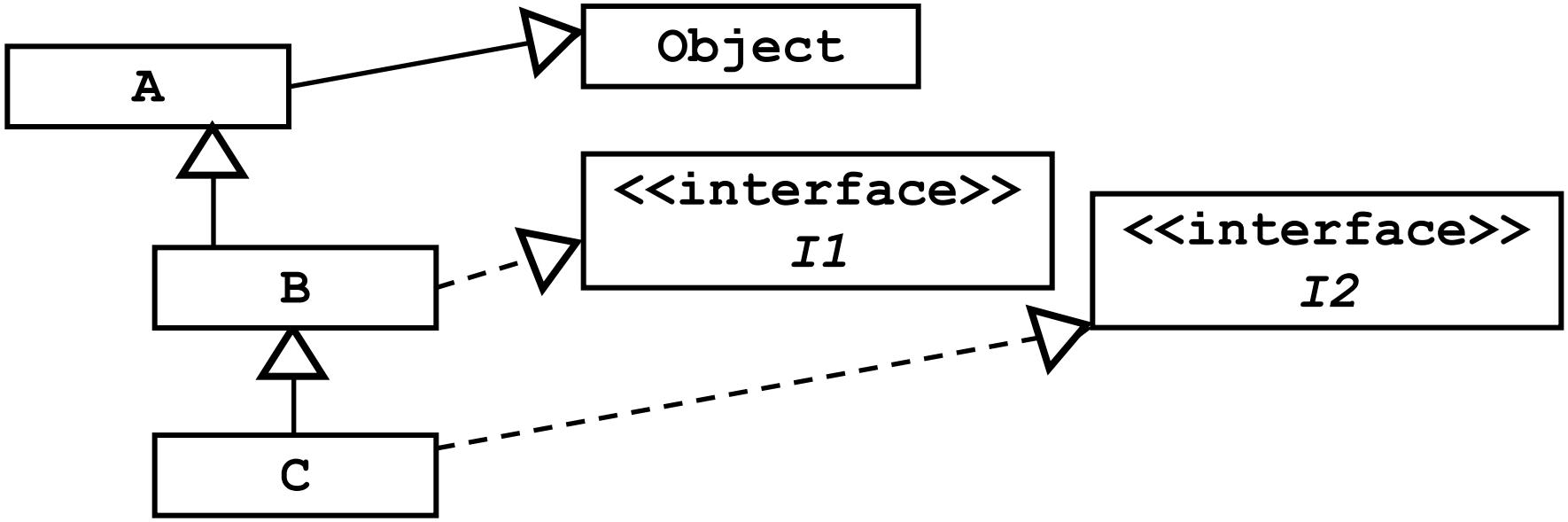


Ereditarietà

- Se una classe estende una superclasse e implementa una o più interfacce ogni suo oggetto istanza si può anche vedere:
  - come istanza della superclasse
  - come istanza di ogni super- ... superclasse
  - tramite il riferimento di una qualsiasi delle interfacce che la sua classe implementa
  - tramite il riferimento a un'interfaccia di cui una di quelle implementate è un'estensione
  - tramite il riferimento a un'interfaccia implementata da una superclasse

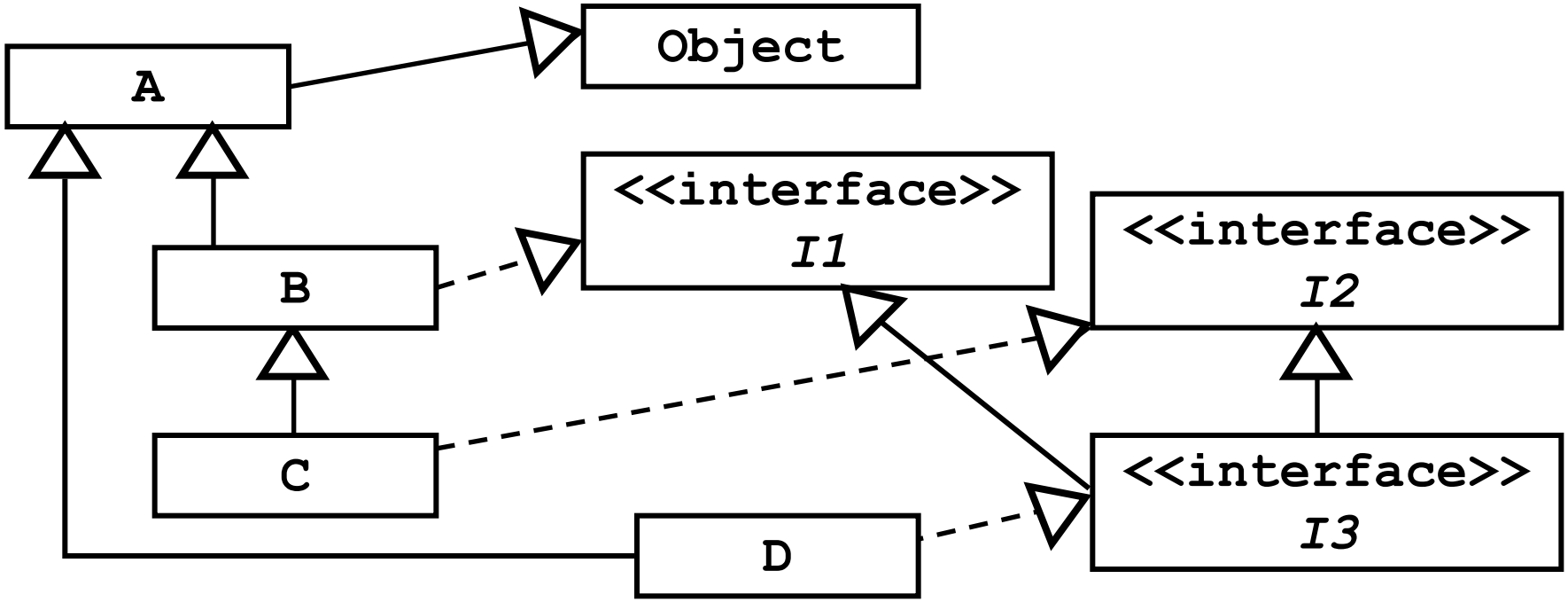
    . . . .

Ereditarietà



- **B** è un sottotipo di **A**. **A** e **I1** sono supertipi di **B**.
- **A**, **B**, **I1** e **I2** sono tutti supertipi di **C**.
- Inseriamo una sottoclasse diretta di **A** detta **D** che implementa un'interfaccia **I3** che estende **I1** e **I2**.

Ereditarietà



- Quali sono i supertipi di **`D`**? **`A`**, **`I1`**, **`I2`** e **`I3`**.
- Quali oggetti possono essere riferiti da una variabile di tipo **`I2`**? Oggetti istanza di **`C`** e **`D`**.

Polimorfismo

- Se definiamo un **`metodo()`** nell'interfaccia **`I2`**, e una variabile riferimento **`xyz`** è dichiarata di tipo **`I2`**, cosa dire di **`xyz.metodo()`**?
  - È un'invocazione lecita secondo il compilatore, perché è un metodo esistente per l'interfaccia
  - Quando viene eseguita, bisogna che la variabile riferisca un oggetto, che può essere istanza della classe **`C`** o **`D`** ma non di **`I2`** che è un'interfaccia.
  - Quello che viene effettivamente eseguito (a run-time) dipende dal **tipo dinamico** dell'oggetto.

Classi astratte vs. interfacce

- Le classi astratte consentono solo ereditarietà singola: → se serve ereditarietà multipla, occorre usare un'interfaccia.
- Le interfacce non hanno implementato alcun metodo, i loro metodi sono tutti pubblici e hanno solo costanti: → se serve qualche metodo già implementato (di istanza o statico) o un membro privato, occorre usare una classe astratta.

Classi astratte vs. interfacce

- In generale le interfacce sono considerate strutture più flessibili.
  - Possono estendere **più di un'altra** interfaccia e possono sempre essere implementate da una classe (eventualmente astratta) mentre non è possibile la conversione inversa.
- Una classe astratta invece può essere estesa da **una sola** altra classe, per il principio dell'ereditarietà singola.

Esempi notevoli

- Per leggere input abbiamo visto la classe **`BufferedReader`**. Questa classe viene costruita a partire da parametri **`Reader`**.
- **`Reader`** è una classe astratta (di cui la stessa **`BufferedReader`** è una sottoclasse concreta)
- A sua volta implementa le interfacce **`Readable`** e **`Closeable`** che significano:
  - che è un oggetto che può essere letto
  - che è un oggetto che può essere chiuso
- E quindi prevedono modi per farlo

La classe `Scanner`

- È una classe utile per leggere input testuali, ma facendo automaticamente e in modo furbo il parsing di tipi primitivi e stringhe.
- Ha costruttore sovraccarico, e le varianti più importanti passano (come input da scandire):
  - una **`String`**
  - un oggetto **`InputStream`** (come **`System.in`**)
  - un oggetto che implementi l'interfaccia **`Readable`**

La classe `Scanner`

- Ad esempio legge interi da input tastiera:

```
Scanner sc = new Scanner(System.in);
if (sc.hasNextInt())
{ int i = sc.nextInt();
}
```

- Oppure legge stringhe come token:

```
Scanner sc = new Scanner("ab ac ad");
while (sc.hasNext())
{ System.out.println(sc.next());
}
```

La classe `Scanner`

- Può usare altri deliminatori invece di spazio

```
String ww = "7 aX 2    aX java";
Scanner scandisce =
    new Scanner(ww).useDelimiter("\\s*aX\\s*");
System.out.println(scandisce.nextInt());
System.out.println(scandisce.nextInt());
System.out.println(scandisce.next());
scandisce.close();
```

- Produce:

```
7
2
java
```

Collezioni

- La gestione degli array da parte di Java offre un costrutto semplice, ma poco potente.
- Limiti degli array:
  - accesso obbligatoriamente tramite indice intero
  - rigidità della struttura
  - necessità di definire in anticipo il numero (massimo) di elementi dell'array
- Questi problemi sono risolti dalle collezioni

Collezioni

- Le collezioni non sono “native” di Java: sono implementate nel Java Collection Framework che fa parte del package **`java.util`**
- Ne esistono vari tipi. In particolare ci sono:
  - **collezioni**: gruppi di oggetti, tra cui distinguiamo **liste** (sequenze) e **insiemi** (collezioni senza duplicati)
  - **mappe**: insiemi di coppie <chiave, campi> in cui i campi contengono i valori significativi, ma vengono acceduti tramite le chiavi

ArrayList

- Una ArrayList (lista sequenziale) è una struttura dati che può memorizzare una sequenza di oggetti.
  - Quindi è simile a un array. Conserva l'ordine degli oggetti secondo come sono stati inseriti.
- Però rispetto a un array:
  - non ha limitazioni alla capacità: non ha numero massimo di elementi, se serve spazio se lo crea
  - sa quanti oggetti contiene (cfr. **`length`**)

ArrayList

- Principali metodi di una ArrayList:
  - **`get(int i)`**: ritorna l'oggetto i-esimo
  - **`set(int i, Object o)`**: imposta l'oggetto i-esimo

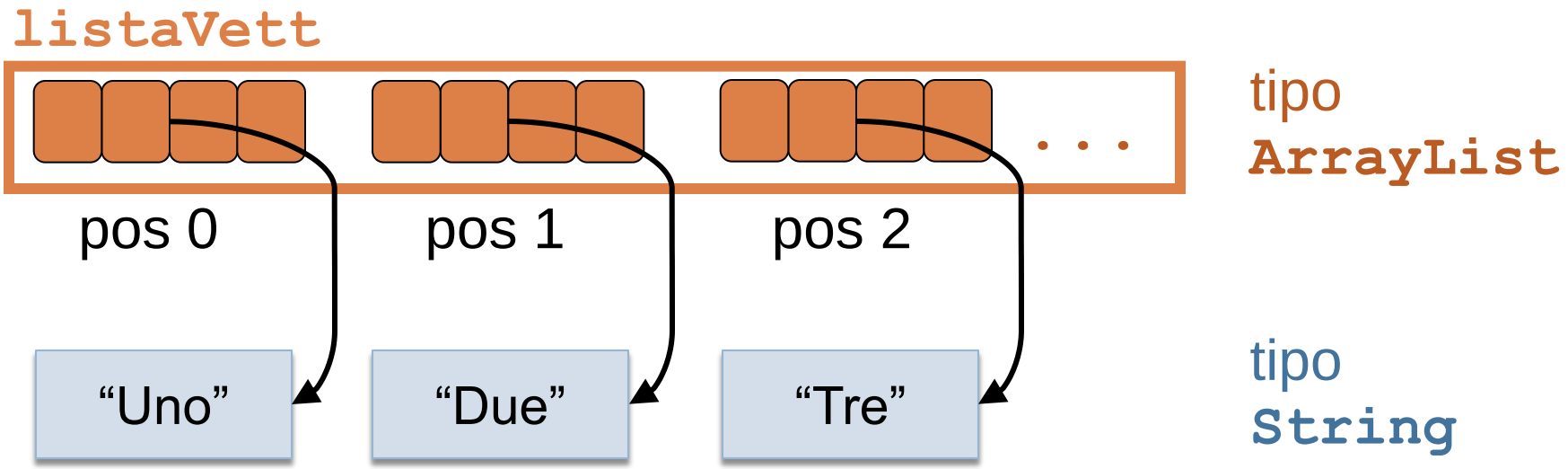
metodi tipici di un array

  - **`add(int i, Object o)`**: aggiunge un oggetto alla lista in posizione i
  - **`add(Object o)`**: aggiunge in fondo
  - **`remove(int i)`**: rimuove l'oggetto in posizione i-esima e lo ritorna
  - **`size()`**: dà il numero di oggetti in lista
  - **`isEmpty()`**: boolean di lista vuota

metodi tipici di una lista

ArrayList

- Gli elementi dell'ArrayList hanno una posizione (implicita) che parte da 0, come per gli array.



ArrayList

```
ArrayList esempio = new ArrayList();
esempio.add("Uno");
esempio.add("Due");
esempio.add("Tre");
for (int i=0; i<esempio.size(); i++)
{ String testo = (String) esempio.get(i);
  ...
}
esempio.set(1, "Sedici");
esempio.remove(0);
```

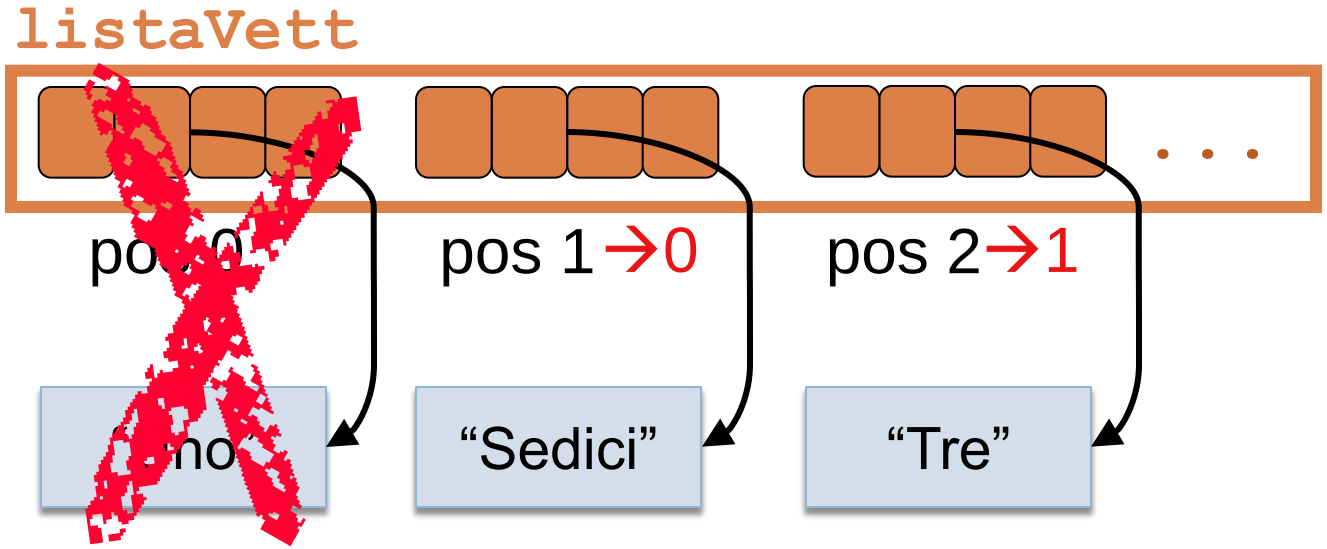
ArrayList

- I metodi **`get()`** e **`remove()`** ritornano un valore di tipo **`Object`**. È necessario un cast al tipo dell'oggetto ritornato.

```
ArrayList esempio = new ArrayList();
esempio.add("Elemento"); // in posizione 0
String elem = (String) esempio.get(0);
esempio.isEmpty(); // vale false
elem = (String) esempio.remove(0);
esempio.isEmpty(); // vale true
```

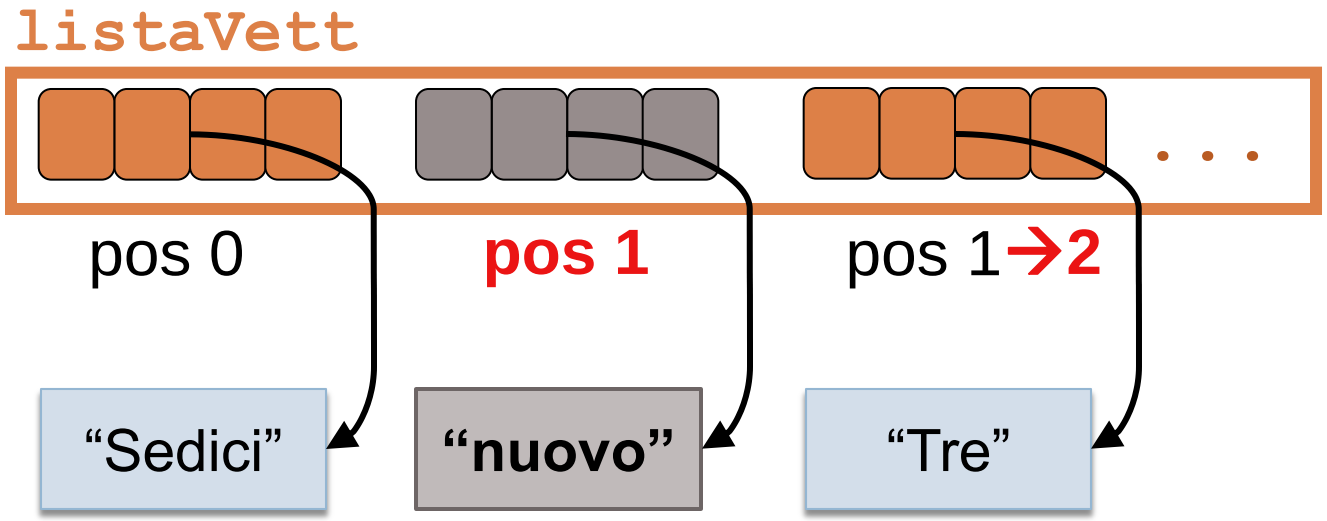
ArrayList

- Una complicazione dovuta alle operazioni di rimozione è che esso cambia gli indici di tutti gli elementi che seguono quello rimosso, portandoli indietro di una unità



ArrayList

- Allo stesso modo, anche l'aggiunta di elementi in lista tramite il metodo **`add()`** sposta in avanti gli elementi (essendo un metodo col nome sovraccarico, esiste anche la variante in cui si può inserire in posizione arbitraria).



LinkedList

- Una lista concatenata (LinkedList) aggiunge allo schema dell'ArrayList una struttura simile alle liste bidirezionali del C
  - ogni elemento contiene un riferimento a quello che segue e quello che precede
- Si può usare una modalità di scansione
  - consente facilmente inserimenti e rimozioni
  - tuttavia rende più difficile l'accesso diretto agli elementi della lista

LinkedList

- Ha tutti i metodi dell'Arraylist:
  - **`get(int i)`**: ritorna l'oggetto i-esimo
  - **`set(int i, Object o)`**: imposta l'oggetto i-esimo

metodi tipici di un array

  - **`add(int i, Object o)`**: aggiunge un oggetto alla lista in posizione i
  - **`add(Object o)`**: aggiunge in fondo
  - **`remove(int i)`**: rimuove l'oggetto in posizione i-esima e lo ritorna
  - **`size()`**: dà il numero di oggetti in lista
  - **`isEmpty()`**: boolean di lista vuota

metodi tipici di una lista

LinkedList

- In aggiunta, la lista concatenata consente anche:
  - **`addFirst, addLast(Object o)`**
  - **`removeFirst, removeLast()`**

metodi tipici di una lista

- Questi metodi sono utili se vogliamo implementare una pila o coda (politiche LIFO / FIFO)

Scansione di una lista

- Una lista (meglio se concatenata) può anche essere scandita:
  - **`listIterator()`**: attivare un **`ListIterator`** che itera sulla lista

iteratore

  - **`next()`**: prossimo elemento della lista
  - **`hasNext()`**, **`nextIndex()`**: c'è un next, e che posizione ha
  - **`previous()`**, **`hasPrevious()`**, **`previousIndex()`**: analoghi al next
  - **`add()`**, **`remove()`**: aggiungi/rimuovi

metodi del **`ListIterator`** per scandire

Iteratore

- L'iteratore è un oggetto che va abbinato ad una lista e serve a scandirla.
- Viene creato dal metodo **`listIterator()`** della lista, come se fosse un costruttore (ma fare attenzione alla minuscola).

```
LinkedList conct = new LinkedList();
ListIterator it = conct.listIterator();
```

crea l'iteratore

crea la lista

Iteratore

- L'iteratore **non è** una posizione della lista.
- Casomai, è intermedio tra due posizioni (di cui una eventualmente è un "bordo")
  - una di esse è la posizione **`next()`**, l'altra è la posizione **`previous()`**
  - se la lista contiene n elementi ci sono n+1 "posti" in cui può stare un iteratore (dal posto prima di 0 a quello dopo n-1)

Iteratore

- Un iteratore ha metodi **`hasNext()`** e **`nextIndex()`** che dicono rispettivamente: se c'è un prossimo elemento e la sua posizione (analoghi per previous)
  - Vanno invocati **sull'iteratore**, non sulla lista!
- Inoltre, l'iteratore ha anch'esso i suoi metodi **`add()`** e **`remove()`** per aggiungere e prelevare oggetti dalla lista.
  - Anche questi sono invocati sull'oggetto iteratore e non sulla lista!

Uso dell'iteratore

- Scansione di una lista con un ciclo:

```
LinkedList esempio = new LinkedList();
esempio.add("Uno"); esempio.add("Due");
...
for (int i=0; i<esempio.size(); i++)
{ String testo = (String) esempio.get(i);
  ...} //versione sequenziale
ListIterator it = esempio.listiterator();
while (it.hasNext())
{ String testo = (String) it.next();
  ...} //versione iteratore
```

Uso dell'iteratore

- Quando si invoca **`next()`** sull'iteratore serve un cast, perché il tipo di ritorno è **`Object`**.
- Si può spostare in questo modo l'iteratore fino a una specifica posizione desiderata.
  - Si può anche (ad esempio se si è scandita tutta la lista) chiedere un nuovo iteratore, e usare solo quello o farli coesistere per scansioni incrociate.
- Se si usano **`previous()`** o **`next()`** con l'iteratore che così "esce dai bordi" viene sollevata una **`NoSuchElementException`**

Uso dell'iteratore

- Il metodo **`add(Object obj)`** dell'iteratore inserisce l'oggetto **`obj`** nel punto dove si trova l'iteratore (cioè prima della posizione di **`next()`** e dopo quella di **`previous()`**).
  - L'iteratore poi resta tra il nuovo elemento e **next**.
- Il metodo **`remove()`** invece rimuove dalla lista l'ultimo oggetto che è risultato come chiamata di un **`next()`** o di un **`previous()`**
  - se ne può invocare solo uno per next/previous
  - non si può chiamare **`remove()`** dopo un **`add()`**

Uso dell'iteratore

```
LinkedList q = new LinkedList();
q.add("A"); q.add("B"); q.add("C");
//si parte con q che contiene ABC

ListIterator it = q.listiterator(); // ^ABC
it.next(); it.next(); // AB^C
it.remove(); // A^C  -- B era l'ultimo next
it.add("F");   // AF^C
it.previous(); // A^FC
it.add("E"); it.add("D"); // AED^FC
it.previous(); // AE^DFC
it.remove(); // AE^FC
it.remove(); // ERRORE! non si può invocare
```

Discussione

- L'iteratore è ereditato dalla classe astratta **`AbstractList`**, di cui sia **`ArrayList`** che **`LinkedList`** sono sottoclassi.
  - Però sulle ArrayList conviene usare direttamente i metodi get e set, invece sulle LinkedList conviene usare un iteratore che le scandisca
  - A livello di rappresentazione, una ArrayList modella un array, mentre una LinkedList è l'analogo di una lista bidirezionale di puntatori.

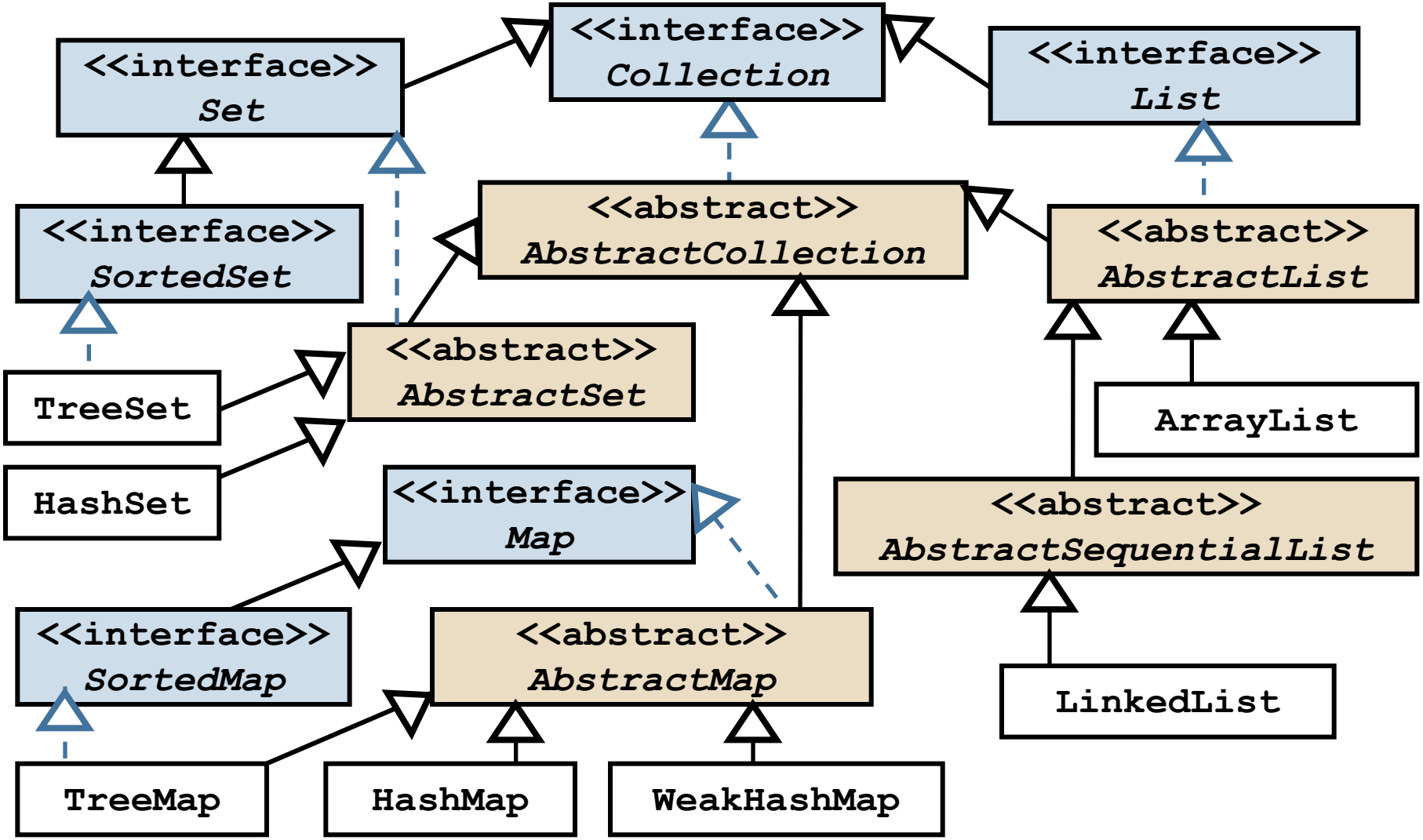
Altre collezioni utili: mappe

- Una mappa (**`Map`**) è un insieme di coppie formate come: (**`Chiave`**, **`Valore`**)
  - Una rubrica telefonica usa come chiave il nome (entry ordinate per nome) e come valore il numero di telefono: è facile trovare un valore corrispondente a una chiave, ma non il viceversa.
- **`Map`** è un'interfaccia. Un suo esempio di implementazione è **`HashMap`** (stile hash table)
  - Metodi principali: **`get`** (ottiene il valore data la chiave) e **`put`** (inserisce una coppia nella mappa)

Altre collezioni utili: insiemi

- L'insieme (**`Set`**) è un'altra interfaccia.
  - In sé rappresenta una lista disordinata: non conta l'ordine di inserimento degli elementi.
  - Inoltre conta solo se un elemento è presente o meno, non ci sono duplicati. Reinserire un elemento presente non ha alcun effetto.
- Esistono diverse implementazioni di questa interfaccia, la maggior parte delle quali sono insiemi con un ordinamento ma dove permane il criterio di non ammettere duplicati.

Schema delle collezioni



Collezioni e interfacce

- Se le interfacce (e anche alcune classi astratte) giocano un ruolo importante nelle collezioni, è possibile sfruttare il polimorfismo.
- Si può usare una variabile riferimento di tipo **`List`** (o **`Set`**, o **`Map`**, che sono interfacce) per un oggetto istanza di una classe più specifica, sfruttando così il polimorfismo.

```
List l = new ArrayList();
```

- È lecita perché **`ArrayList`** implementa **`List`**

Collezioni e tipi generici

- Caratteristica importante delle collezioni è che si può memorizzare **qualunque tipo** di oggetti (diversamente da quanto avviene per gli array)
- In linea di principio gli oggetti collezionati sono di tipo **`Object`**.
  - Si può sempre quindi inserire un oggetto qualsiasi nella collezione.
  - Quando lo si estrae invece è necessario un cast per avere il tipo desiderato: i metodi per estrarlo ritornano un riferimento a **`Object`**

Collezioni e tipi generici

- Dalla versione 5 di Java è però possibile definire un tipo generico per le collezioni.
- Ogni volta che si crea una collezione si può specificare un tipo degli oggetti che ci si attende siano contenuti nella collezione.
- Va specificato tra parentesi angolari **< >**.

```
List<String> l = new ArrayList<String>();
```

- In questo modo non è necessario specificare per ogni oggetto che si estrae che è **`String`**.

Collezioni e tipi generici

```
List x = new ArrayList();
x.add("Uno"); x.add("Due");
String s1 = (String) x.get(0);
Studente s2 = (Studente) x.get(1);
```

istruzione sbagliata: infatti **`Studente`** non è né **`String`** né una sua superclasse. Tuttavia il compilatore non se ne accorge perché accetta il cast esplicito da **`Object`** a **`Student`**. Ci sarà un errore a run-time (eccezione)

Collezioni e tipi generici

```
List<String> x = new ArrayList<String>();
x.add("Uno"); x.add("Due");
String s1 = (String) x.get(0);
Studente s2 = (Studente) x.get(1);
```

l'istruzione è sempre sbagliata ma stavolta il compilatore se ne accorge perché si aspetta un tipo **`String`** e non **`Studente`**