Programma Arnaldo 2020

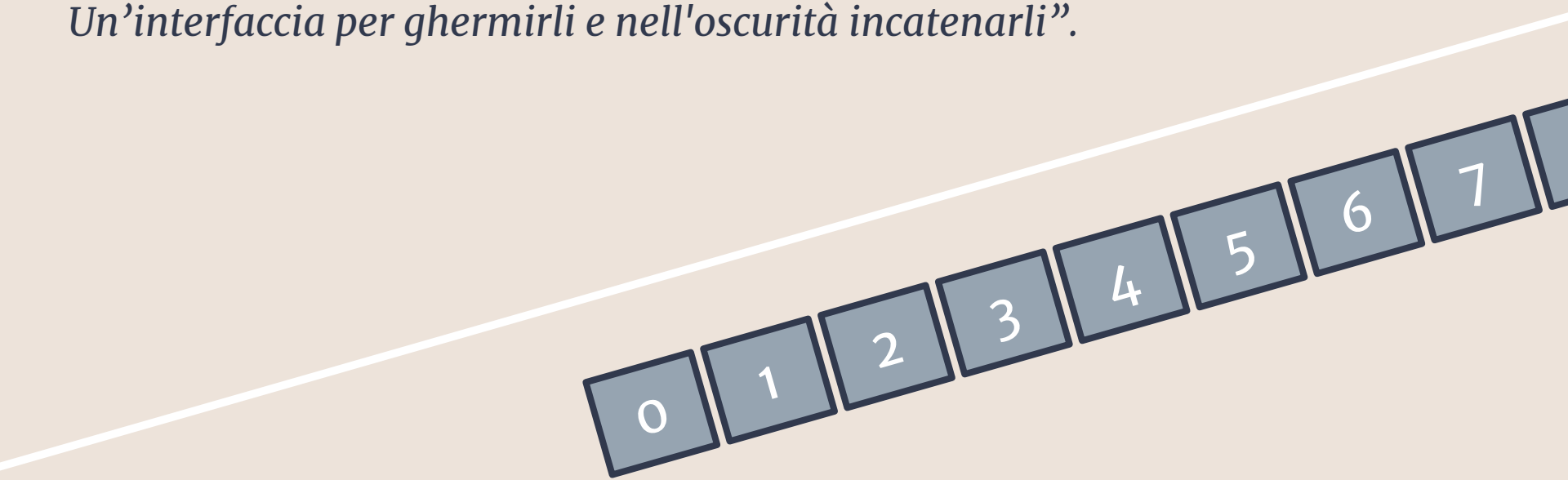
Liste

nel JCF

Lezione 1.2

L’interfaccia List

“Un’interfaccia per domarli, un’interfaccia per trovarli,
Un’interfaccia per ghermirli e nell'oscurità incatenarli”.



Perchè non gli Array?

In Java, come in C, è possibile creare *array* unicamente con **dimensione fissa**.

Esempio: `Pizza[] array_di_pizze = new Pizza[23];`

Ma all'interno del fantastico e bellissimo *Java Collections Framework*, non esiste una classe **array** già pronta?

In realtà, in Java esistono due classi chiamate **Array** e **Arrays**, ma possiedono solo metodi statici e pertanto, solitamente, non vengono istanziate.

Array
Fornisce metodi statici per creare e accedere dinamicamente agli array.

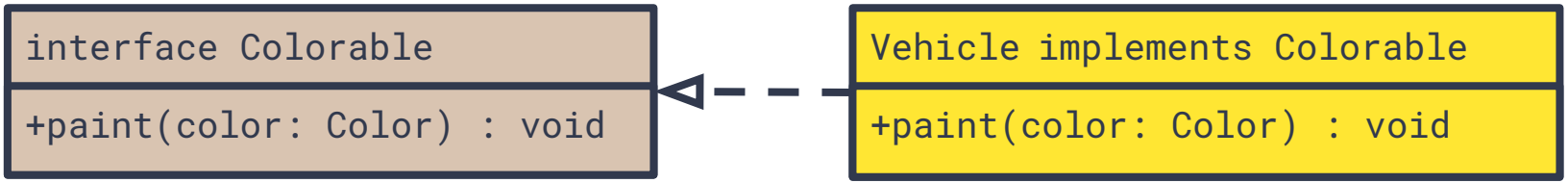
Arrays
Fornisce metodi statici per manipolare un array (ordinamento, ricerca, etc...).

Di cosa abbiamo bisogno?

Vogliamo la rappresentazione di una lista, che sia *istanziabile* come oggetto e che provveda già all'implementazione di alcuni *metodi utili* o "standard".

Interfacce: come trattarle?

List, come già detto, è un'**interfaccia**.
Potete immaginare un'interfaccia come un "comportamento": quando una classe implementa un'interfaccia, sta assumendo anche il comportamento rappresentato da tale interfaccia.
Esempio:

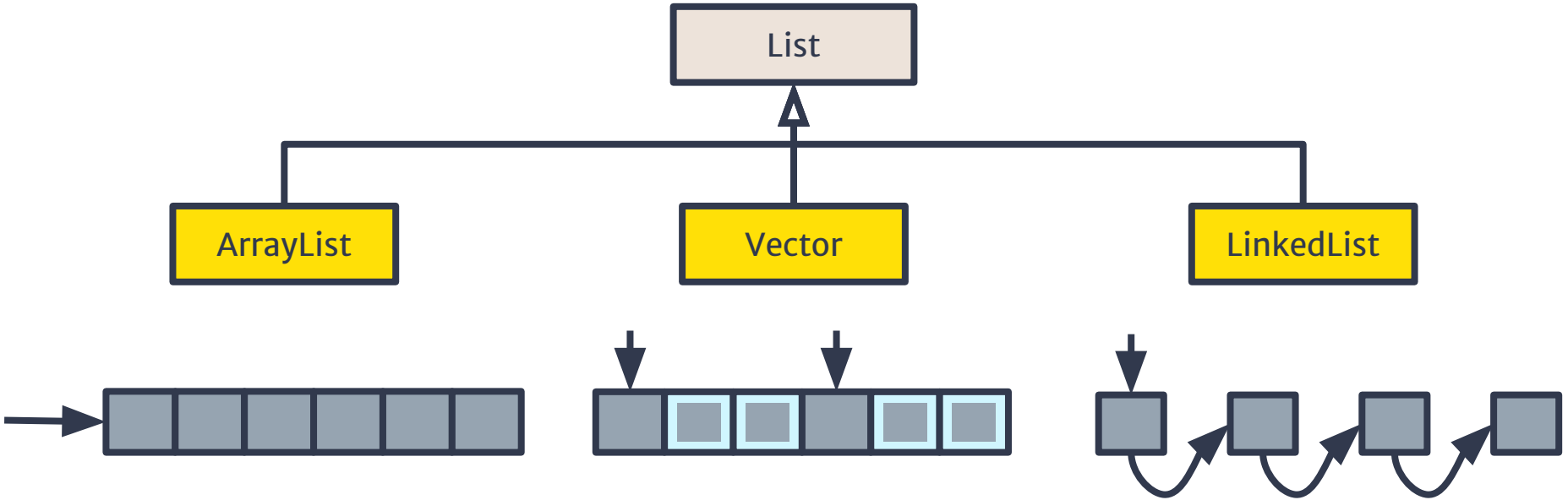


Perchè una classe dovrebbe **implementare** un'interfaccia?

Perchè le interfacce **non sono istanziabili** (di norma – vedi *Risorse* alla fine).

Implementazione di *List* [1]

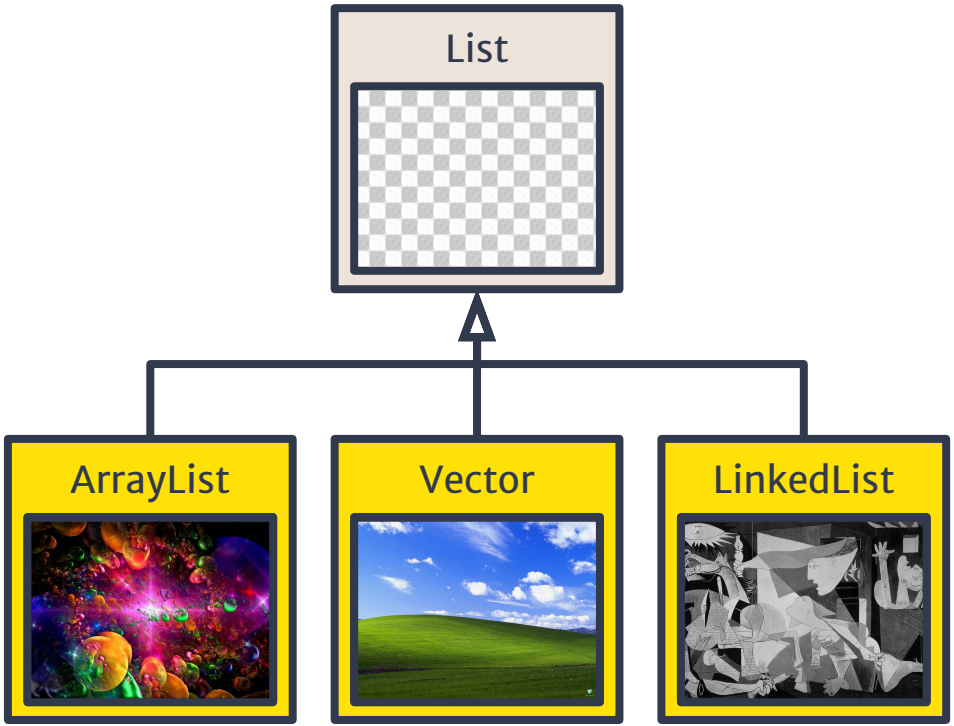
Il *Java Collections Framework* fornisce già tre implementazioni dell'interfaccia *List*, che possono essere (e sono) utilizzate ogni qualvolta sia necessario accorpare oggetti in forma di "lista":



Implementazione di *List* [2]

Come interfaccia, *List* detta un comportamento a tutte le classi che la implementano, le quali però sono libere di perseguire tale comportamento nella maniera che preferiscono.

È come se *List* fornisse una cornice da utilizzare per comunicare in maniera corretta con il mondo esterno, ma ogni classe che implementa *List* è libera di riempire tale cornice di quello che preferisce (a patto che funzioni!).



Passato e futuro

Riassunto fin qui – Sappiamo che:

- *List* nasce per rappresentare una qualunque collezione di oggetti che abbia la configurazione di "lista";
- *List* soddisfa tutte le richieste che potremmo voler fare ad una collezione di tipo lista (aggiunta e rimozione dinamica di elementi, indicizzazione, ricerca, etc...);
- *List* è implementata da tre classi differenti, all'interno del *Java Collections Framework*.

Piano d'attacco:

1. Studiamo come istanziare *List* e quali metodi mette a disposizione, in modo da capire in un colpo solo come lavorare con *ArrayList*, *Vector* e *LinkedList*.
2. Analizziamo le singole implementazioni di *List*, allo scopo di:
   a. capirne punti forti e deboli;
   b. conoscere eventuali casi d'uso da manuale.

Domanda 1.2.1 *[Istanziazione]*

Dato il frammento di codice qui sotto, quale o quali istruzioni possono essere inserite al posto della prima riga per rendere il frammento funzionante?

```
//_____?_____//
elenco.add(new Player());                        // Aggiunge un nuovo giocatore
System.out.printf("Size: %d", elenco.size());   // Stampa la dimensione
```

- ✖ **A)** `Collection elenco = new Collection();`
- ✖ **B)** `Collection elenco = new List();`
- ✖ **C)** `List elenco = new List();`
- ✖ **D)** `List elenco = new ArrayList();`
- ✖ **E)** `ArrayList elenco = new ArrayList();`
- ○ **F)** `List<Player> elenco = new ArrayList<Player>();`
- ○ **G)** `ArrayList<Player> elenco = new ArrayList<Player>();`

A, B, C -> Non è possibile istanziare un'interfaccia.

D, E -> Manca la specificazione del tipo dell'elemento.

Liste “tipizzate”

Quando si osserva la documentazione dell’interfaccia *List* di Java, si nota che la dichiarazione dell’interfaccia viene indicata con `List<E>`. Senza entrare nel merito della notazione o del suo significato, all’atto pratico significa che:

Per creare un oggetto che implementi l’interfaccia *List*, è necessario **specificare** fra parentesi angolari la **classe degli elementi** contenuti in tale lista.

Questa operazione permette di segnalare alla *Java Virtual Machine* il tipo (o meglio, la classe) degli elementi della lista, ossia permette di capire che **tipologia di oggetti** contiene la lista.

Ciò fornisce numerosi vantaggi, anche a livello di comodità nella scrittura del codice; per maggiori informazioni, si vedano i link all’argomento “*Java Generics*”.

Esempi di liste “tipizzate”

```
ArrayList<Giocatore> elenco_giocatori = new ArrayList<Giocatore>();
```

```
List<Utente> amici = new Vector<Utente>();
```

```
LinkedList<Ricetta> menu = new LinkedList<Ricetta>();
```

```
List<List<Casella>> scacchiera = new Vector<List<Casella>>();
for (int i = 0; i < N; i++) {
    scacchiera.add(new ArrayList<Casella>());
};
```

Controesempi di liste “tipizzate”

Esempi di codice **non funzionante**:

```
ArrayList<Pizze> elenco_pizze = new LinkedList<Pizze>();
```

LinkedList **non** è una sottoclasse di *ArrayList*: le due classi implementano la stessa interfaccia, ma non sono legate in modo diretto.

```
ArrayList<int> successione_fibonacci = new ArrayList<int>();
```

Non è permesso istanziare *ArrayList*, *Vector* o *LinkedList* che abbiano come elementi dei tipi primitivi. Per risolvere questo problema, Java mette a disposizione delle **classi wrapper**:
`Integer`, `Long`, `Character`, `Double`, `Float`, `Boolean`, `...`

Esempio di utilizzo base delle liste

```java
// creazione della lista
ArrayList<String> listaSpesa = new ArrayList<String>();

// aggiunta di elementi
listaSpesa.add("Cipolle di Tropea");
listaSpesa.add("Pane");
listaSpesa.add("Broccoli");
listaSpesa.add("Cose che non mi servono ma costano poco");

// rimozione di elementi
for (int i = 0; i < listaSpesa.size(); i++)
   if (listaSpesa.get(i).contains("non mi servono")) {
       listaSpesa.remove(i);
       i--;
   }

// stampa
for (String elemento: listaSpesa)
   System.out.println(elemento);
```

Liste di tipi primitivi

Le strutture dati presenti nel JCF hanno un unico requisito sugli oggetti che possono contenere: che siano, per l'appunto, oggetti.

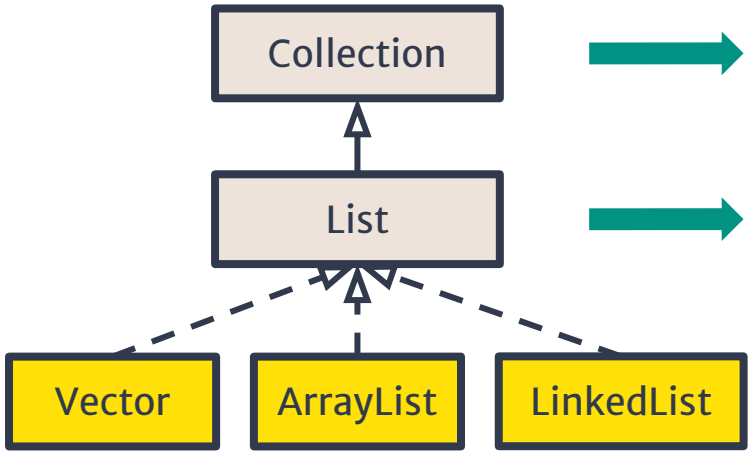
Per creare **liste di tipi primitivi** (int, double, boolean, char, ecc.), è quindi necessario ricorrere alle **classi wrapper**: Integer, Double, Boolean, Character, ecc.

In questo ci viene in aiuto l'**autoboxing** implementato da Java, che si occupa per noi di convertire i tipi primitivi in oggetti e viceversa, in modo trasparente.

```
ArrayList<Integer> lista = new ArrayList<Integer>();
lista.add(3456);  // autoboxing: int -> Integer
int a = lista.get(0);  // unboxing: Integer -> int
System.out.println("a = " + a);                    // a = 3456
System.out.println("lista[0] = " + lista.get(0));  // lista[0] = 3456
```

I metodi di *List*

Per i principi dell'ereditarietà, qualunque classe che implementi l'interfaccia *List* si ritrova ad avere anche tutti i metodi delle interfacce che stanno "sopra" a *List*; pertanto, le classi `ArrayList`, `LinkedList` e `Vector` possiedono anche i metodi dell'interfaccia *Collection*.



Metodi di **utilità generale**, che ritroviamo in una generica struttura dati (anche non lista).

Metodi **propri di una lista**, ad esempio, quelli che avevamo elencato all'inizio:

- indicizzazione;
- ordinamento;
- ricerca;
- ...

Metodi generali

- `size() : int`
  Restituisce il numero di elementi.
- `isEmpty() : boolean`
  Restituisce *true* se la struttura è vuota.
- `toArray() : Object[]`
  Restituisce un array con gli stessi elementi della lista.
- `static shuffle(elements : List<?>) : void`
  Mischia gli elementi della lista; è un metodo statico della classe `Collections` (e non dell'interfaccia *Collection*).
- `sort(comparator : Comparator<? super T>) : void`
  Ordina gli elementi secondo un comparatore.

Aggiunta di elementi

- `add(element : E) : boolean`
  Aggiunge in coda l'elemento (di tipo E); se la struttura viene modificata restituisce *true*.
- `add(index : int, element : E) : void`
  Aggiunge l'elemento di tipo E alla posizione indicata.
- `addAll(c : Collection<? extends E>) : boolean`
  Aggiunge tutti gli elementi della lista c in coda alla lista su cui viene chiamato il metodo.
- `addAll(index : int, c : Collection<? extends E>) : void`
  `Aggiunge tutti gli elementi di c alla posizione desiderata.`

Rimozione di elementi

- `remove(index : int) : E`
  Rimuove l'elemento alla posizione indicata dall'indice e lo restituisce.
- `remove(obj : Object) : boolean`
  Rimuove l'elemento passato come parametro se presente (e in tal caso restituisce true).
- `removeAll(c : Collection<?>) : boolean`
  Rimuove tutti gli elementi della lista c dalla lista corrente.
- `removeIf(Predicate<? super E> filter) : boolean`
  Rimuove tutti gli elementi che soddisfano la condizione di "filtro". Se almeno una rimozione va a buon fine, restituisce *true*.
- `retainAll(c : Collection<?>) : boolean`
  Rimuove tutti gli elementi dalla lista corrente che non appartengono alla collezione c; restituisce true se la lista cambia al termine dell'operazione.
- `clear() : void`
  Svuota la lista.

Accesso e ricerca di elementi

- `get(index : int) : E`
  Restituisce l'elemento alla posizione indicata.
- `set(index : int, element : E) : E`
  Restituisce l'elemento alla posizione indicata e lo sostituisce con l'elemento nuovo passato come parametro.
- `contains(obj : Object) : boolean`
  Restituisce *true* se la lista contiene l'elemento indicato.
- `containsAll(c : Collection) : boolean`
  Restituisce true se la lista contiene tutti gli elementi della collection c.
- `indexOf(element : Object) : int`
  Restituisce la posizione della prima occorrenza dell'elemento, o `-1` se non presente.
- `lastIndexOf(element : Object) : int`
  Restituisce la posizione dell'ultima occorrenza dell'elemento, o `-1` se non presente.

Domanda 1.2.2 *[Esempi di utilizzo]*

Che cosa stampa a schermo il seguente codice?

```
ArrayList<String> parole = new ArrayList<String>();
parole.add("Cumino"); parole.add("Guascone");
parole.add("Pastrano"); parole.add("Frollatura");

String parola_prescelta = new String("Bonobo");

if (parole.addAll(parole)) {
    parola_prescelta = parole.get(parole.size() / 3);
} else {
    parole.add(parole.size() / 2, parola_prescelta);
}

parole.sort(String::compareTo); // Ordina alfabeticamente le parole.
System.out.println(parole.indexOf(parola_prescelta));
```

Domanda 1.2.1 *[Esempi di utilizzo]*

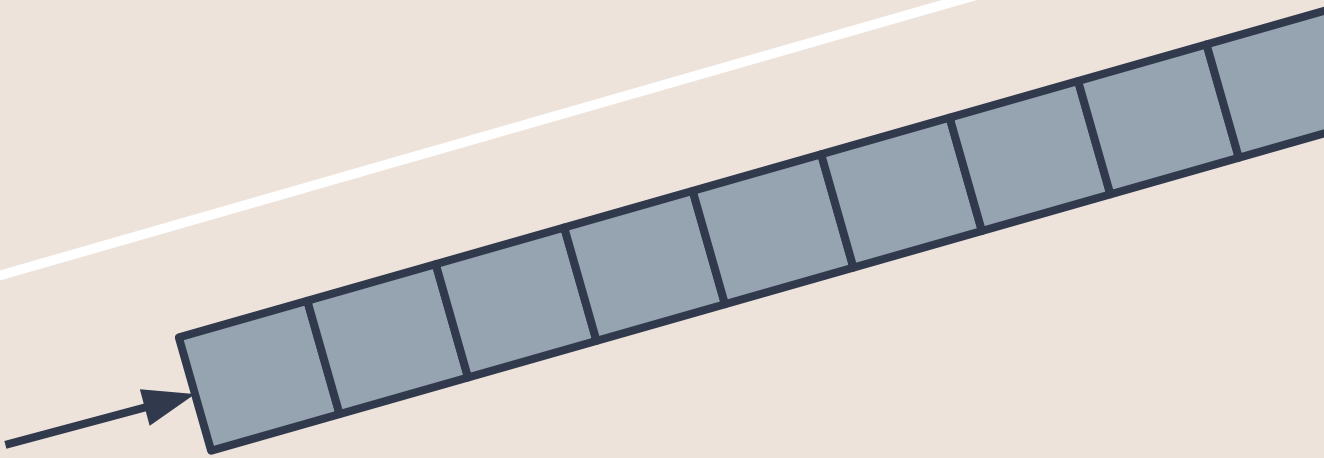
Che cosa stampa a schermo il seguente codice?

```
LinkedList<Integer> elencoA = new LinkedList<Integer>();
LinkedList<Integer> elencoB = new LinkedList<Integer>();
for (int i = 0; i < 10; i++) {
    if (i % 3 == 1) {
        elencoA.add(i);
    }
    else if (i % 4 != 0) {
        elencoB.add(i + 1);
    }
}
elencoA.retainAll(elencoB);
System.out.println(elencoA.size());
```

ArrayList

“Un ArrayList è un array
che non ha mai smesso di sognare.”

- *Bob Dylan*



Cosa c’è sotto?

Un oggetto `ArrayList<E>` è un oggetto che maschera al suo interno un array, ossia che contiene come attributo un array di elementi di tipo E.

```
ArrayList<E> implements List<E>
```

```
- array : E[]
- size : int
...
```

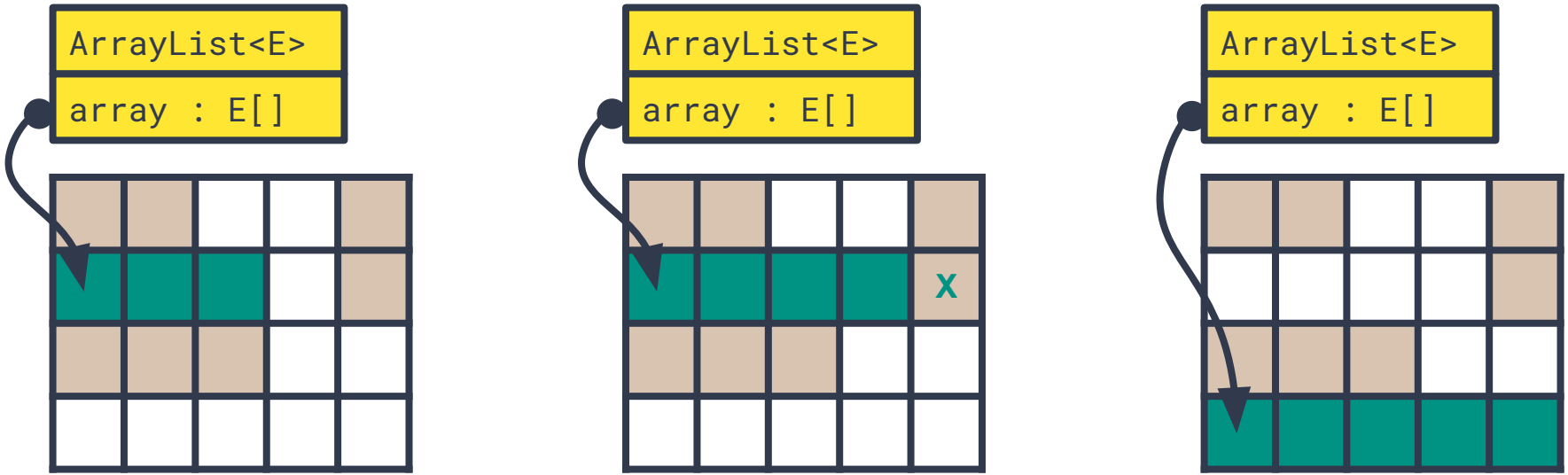
```
+ add(element : E) : void
+ get(index : int) : E
...
```

Problema: gli array hanno dimensione fissa.

Come può un oggetto di tipo ArrayList modificare la dimensione del proprio array in maniera **dinamica**?

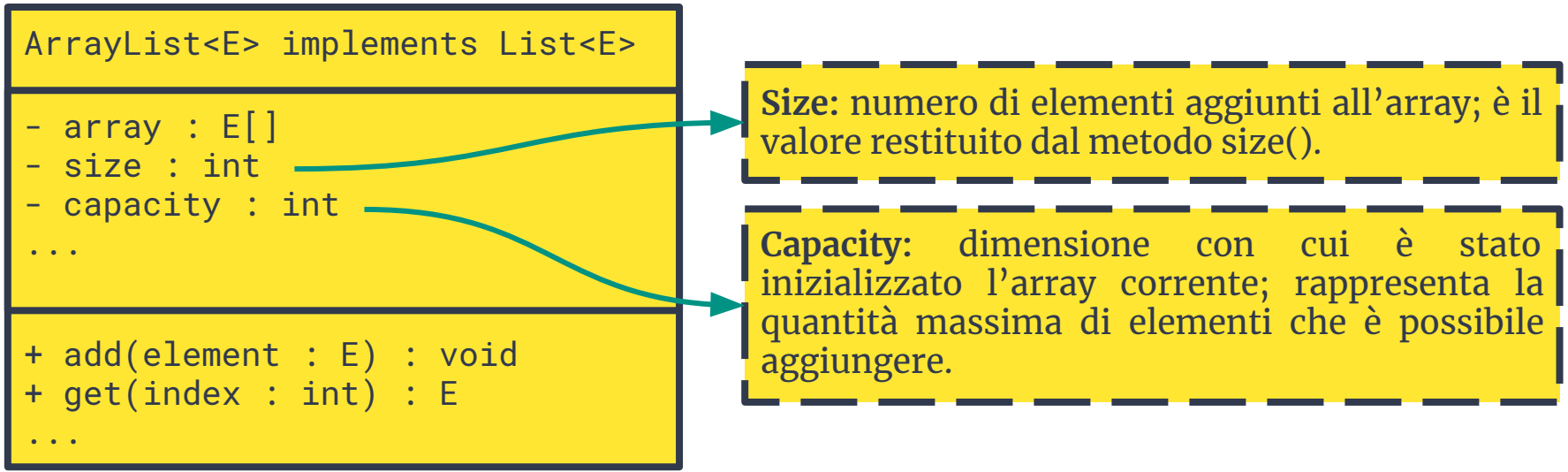
Dimensione dinamica: come funziona?

Un *ArrayList* può modificare la dimensione del proprio array interno con operazioni di **riallocazione della memoria**.



Size *vs* Capacity

Un *ArrayList* capisce quando effettuare operazioni di riallocazione grazie ai valori di due suoi attributi: `size` e `capacity`.



Quando riallocare?

Un *ArrayList* effettua i**n automatico** operazioni di riallocazione (sia per dimensioni maggiori, sia per dimensioni inferiori a quella corrente) confrontando i valori degli attributi `size` e `capacity`.

Il codice, nei metodi che aggiungono elementi, sarà qualcosa simile a:

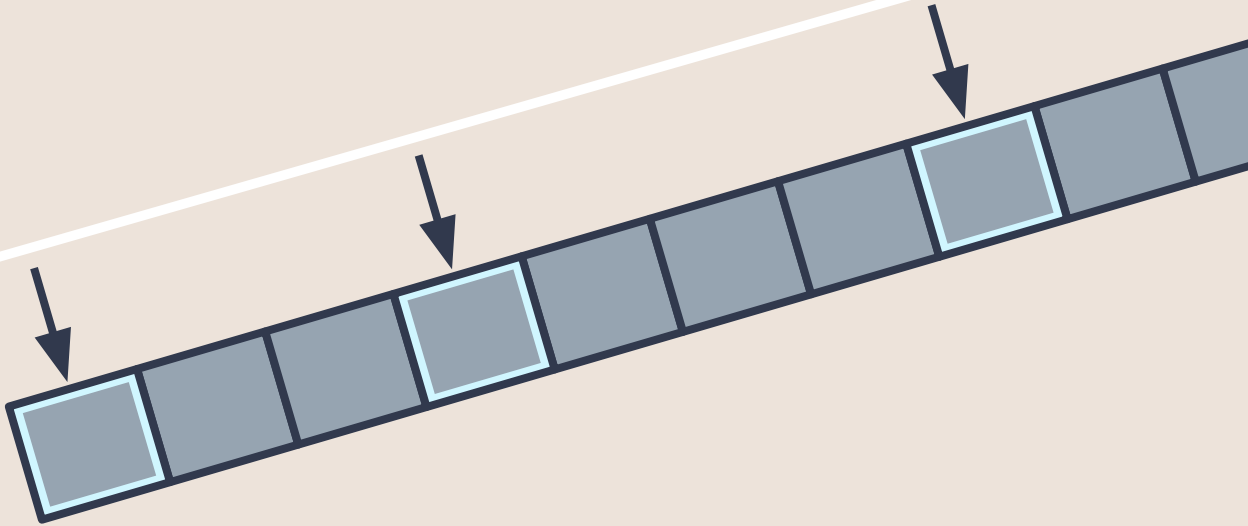
```
if (this.size > [una_certa_percentuale] * this.capacity) {
    this.reallocate(...);     // Funzione di esempio
}
```

È possibile anche aumentare manualmente la dimensione di un ArrayList con la funzione `ensureCapacity(minCapacity : int) : void` che "prepara" l'array ad accettare almeno *[minCapacity]* elementi.

Vector

“Uno alla volta, per carità!”

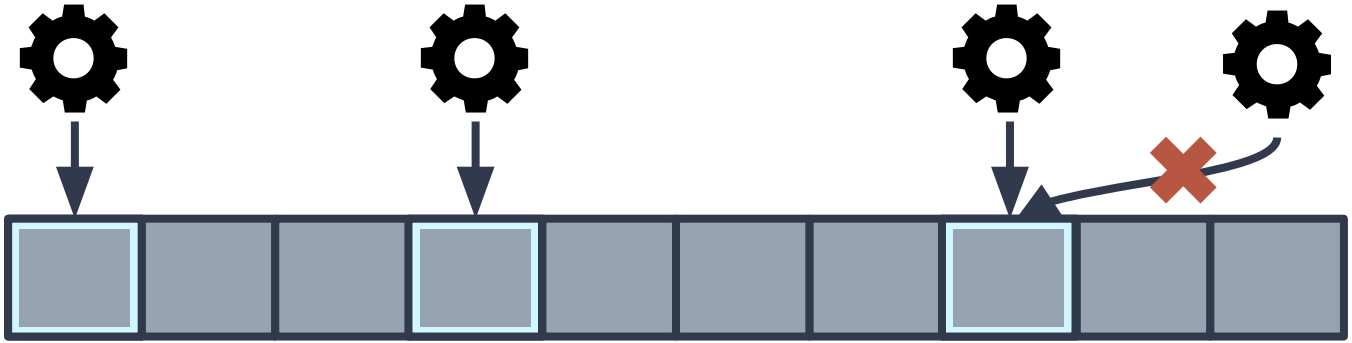
- *Il Programmatore di Siviglia*



Perchè un Vector?

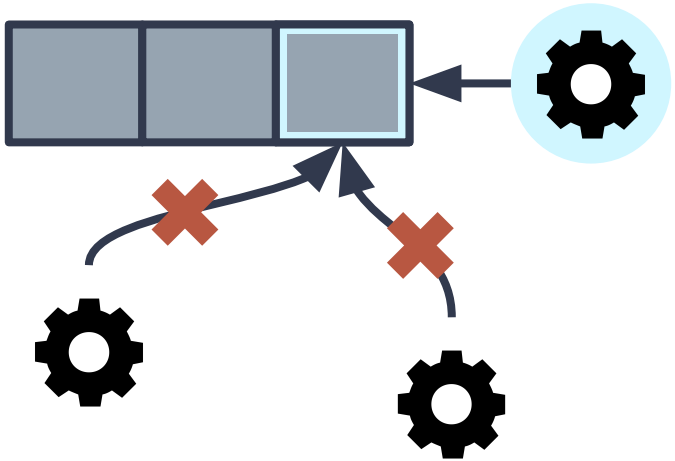
La classe *Vector* è stata creata per lavorare allo stesso modo della classe *ArrayList*, ma in ambiti dove fosse necessario il supporto alla programmazione **multi-threading**. Anche la classe *Vector*, infatti, contiene un array che può essere riallocato a seconda dei casi. Inoltre:

Un oggetto `Vector` permette l'accesso ad un elemento dell'array ad **un solo thread alla volta**.



Come funziona?

Come succede comunemente nella programmazione *multi-threading*, un *Vector* protegge i propri elementi dall'accesso multiplo utilizzando una specie di "**scudo**": tale scudo viene applicato da un thread quando accede all'elemento e viene rimosso al termine dell'operazione.



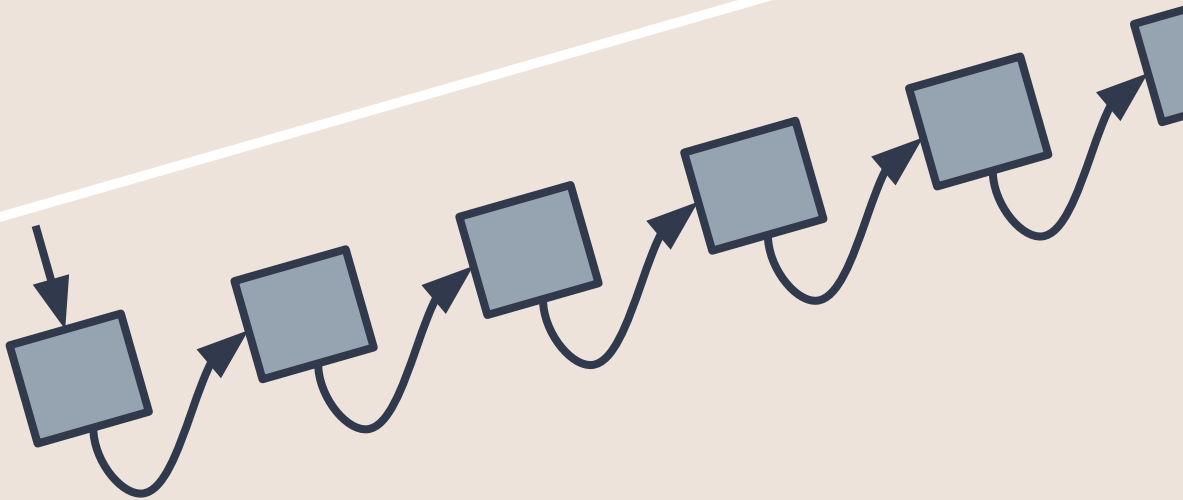
Finchè lo scudo è applicato, nessun altro thread può modificare l'elemento.

Questo risolve molti problemi da un lato, ma **riduce enormemente l'efficienza** in un contesto in cui la programmazione *multi-threading* è assente.

Per questo motivo, noi non utilizzeremo la classe *Vector*, che verrà quindi sostituita quasi completamente dalla classe *ArrayList*.

LinkedList

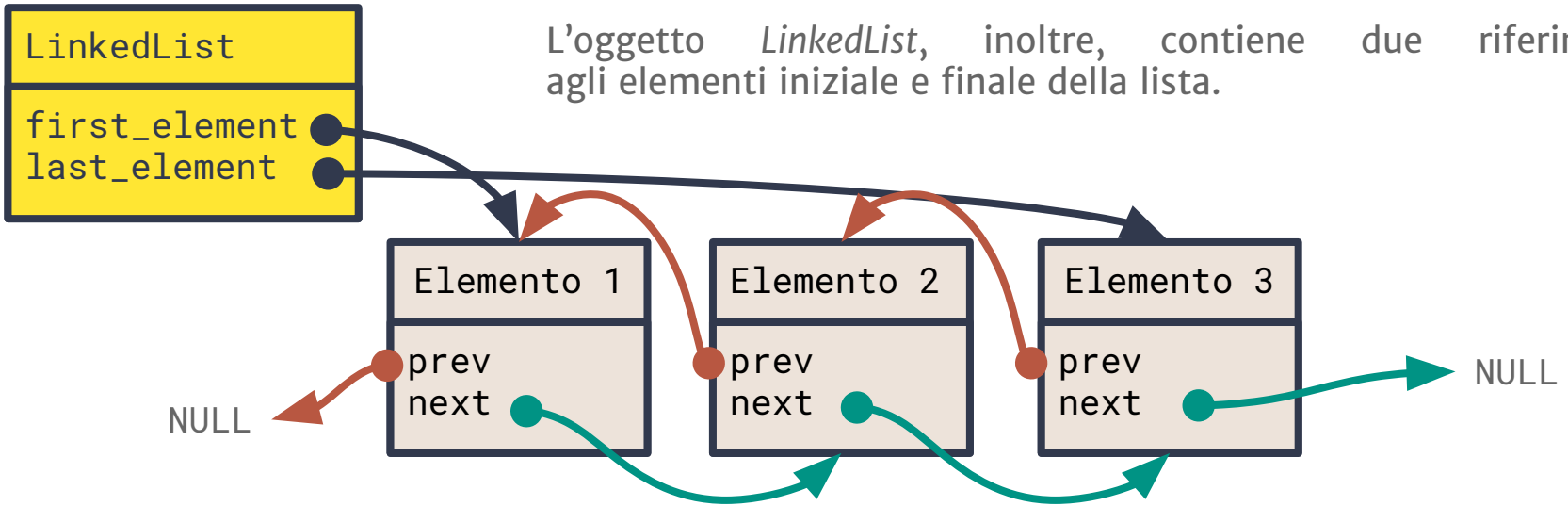
“Non è importante dove sei,
ma chi hai vicino a te.”
- *Gandhi*



Liste Concatenate: cosa sono?

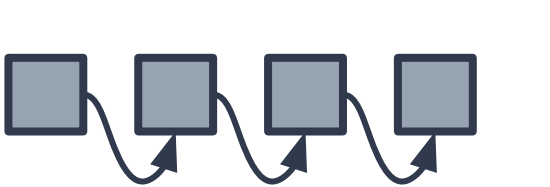
Una *LinkedList* in Java presenta le stesse funzioni di un *ArrayList*, ma è implementata in modo completamente differente: **ogni elemento della lista è incapsulato in un nodo**, che presenta dei riferimenti ai nodi precedenti e successivi.

L'oggetto *LinkedList*, inoltre, contiene due riferimenti agli elementi iniziale e finale della lista.

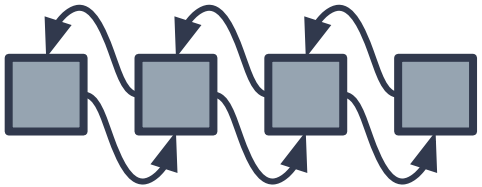


Liste concatenate: un po’ di teoria

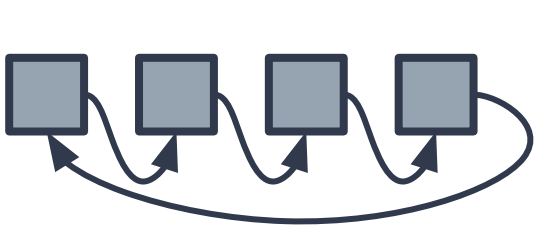
A seconda della configurazione assunta dalla struttura, e dai collegamenti contenuti in ciascun nodo, è possibile **classificare** differentemente le varie liste concatenate:



Liste concatenate **semplici**, o concatenate **singolarmente**



Liste **doppiamente** concatenate

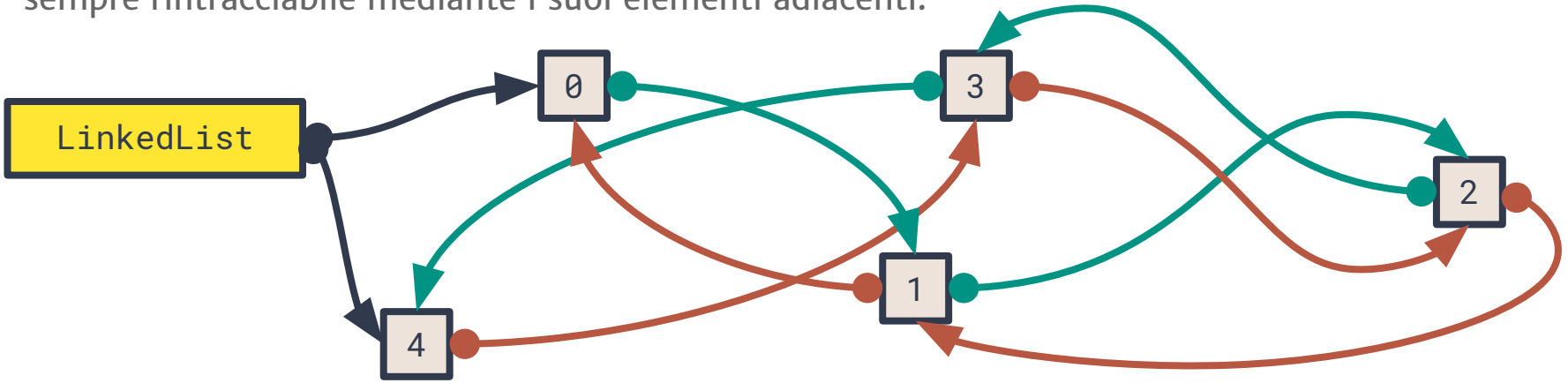


Liste concatenate **ad anello**

Spesso, per implementare una lista con dim. dinamica in C, si modellizza una lista concatenata.

Come funzionano?

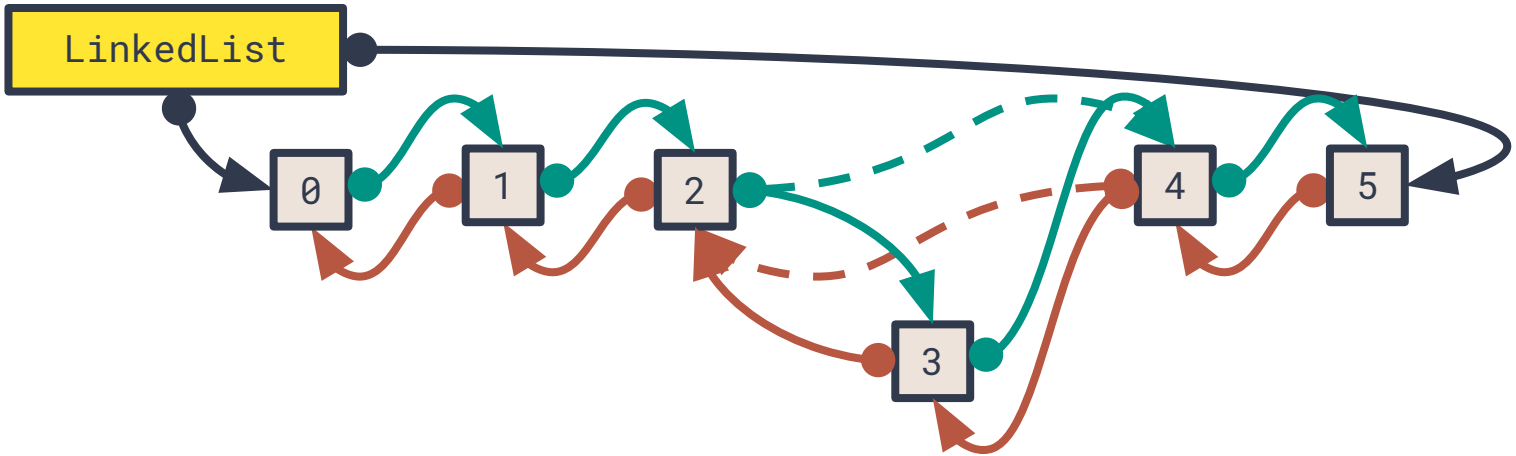
Ogni nodo della lista può essere salvato in memoria in una qualunque posizione, poiché è sempre rintracciabile mediante i suoi elementi adiacenti.



Per questo motivo, recuperare un elemento ad un dato indice all'interno di una lista concatenata richiede lo **scorrimento dell'intera lista** fino al particolare elemento.

Aggiunta e rimozione

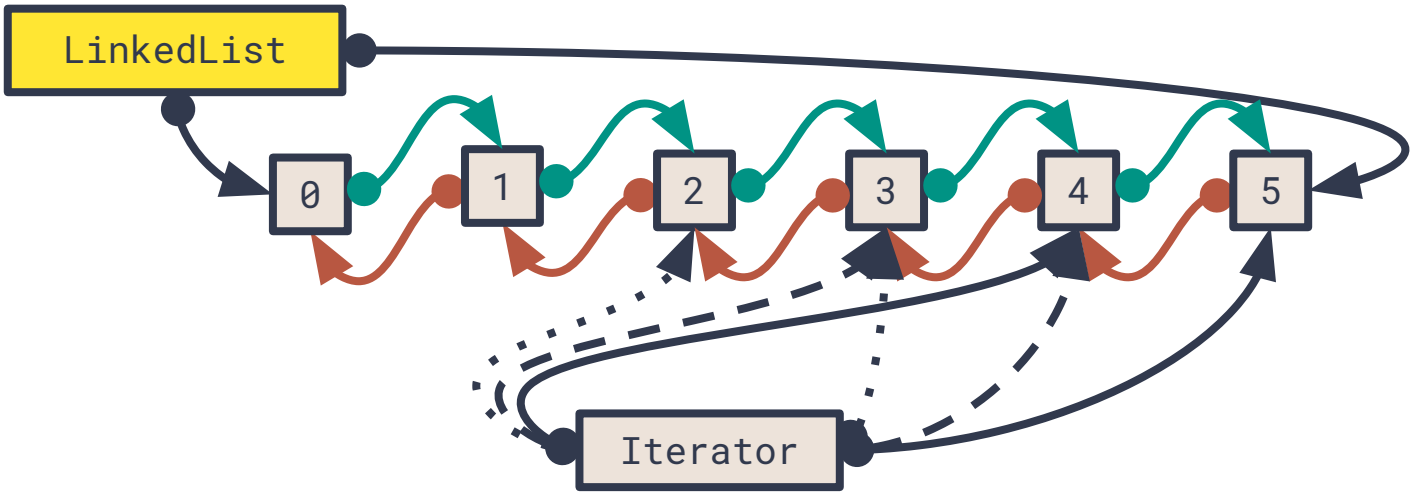
L'aggiunta e la rimozione di elementi in centro ad una lista concatenata sono operazioni particolarmente **efficienti**, poiché richiedono solamente **l'aggiornamento dei collegamenti** fra i nodi.



Ovviamente, la posizione di aggiunta o di rimozione della lista deve essere prima "raggiunta" da un qualche riferimento (spesso tale funzione è svolta da un `Iterator`).

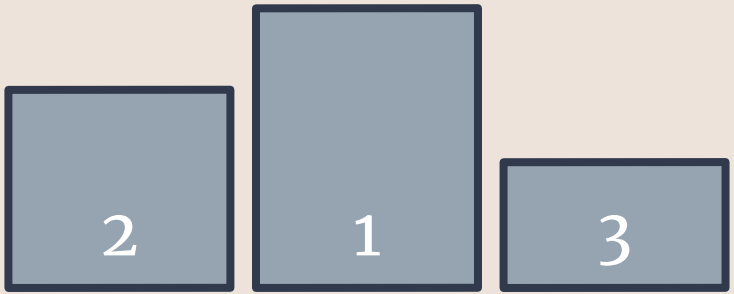
Iterator

Un **iterator** è un oggetto che permette di scorrere agevolmente una lista, tenendo traccia tramite riferimenti dell'elemento successivo e di quello precedente.



Implementazioni a confronto

“Quindi qual è il migliore?”
“Dipende.”
- *antico proverbio zen*



Pro e Contro

Una diversa implementazione interna delle classi porta ad una differenza di efficienza nelle operazioni che tali classi possono fare. Ad esempio, un *ArrayList* sarà favorito dalla "compattezza" dei suoi dati, mentre una *LinkedList* trarrà vantaggio dalla libertà con cui sono collegati i suoi elementi.

Più nello specifico, possiamo dire che:

Un **ArrayList** è utile quando si può sfruttare l'accesso casuale, ossia l'accesso ad un qualunque elemento in maniera "diretta". Non è consigliato per frequenti aggiunte o rimozioni in mezzo alla lista.

Una **LinkedList** è utile quando si deve aggiungere o rimuovere un elemento in mezzo o agli estremi della lista. Non è comoda quando si deve accedere ad un elemento in base al suo indice.

Efficienza algoritmica dei metodi

	ArrayList	LinkedList
Aggiunta o rimozione di un elemento in testa	O(n)	O(1)
Aggiunta o rimozione di un elemento in coda	O(1)	O(1)
Aggiunta o rimozione di un elemento “in mezzo”	O(n)	O(1)
Lettura o modifica di un elemento agli estremi	O(1)	O(1)
Lettura o modifica di un elemento “in mezzo”	O(1)	O(n)
Concatenazione di due liste	O(n)	O(1)
Attraversamento dell’intera lista	O(n)	O(n)

Appendice

Esempi e casi d'uso

Esempio 1.2.2

Ordinamento di Stringhe
Il frammento di programma presentato nelle successive slide mostra un esempio di utilizzo della classe **LinkedList** per contenere una sequenza di stringhe.
Tali stringhe vengono lette da tastiera mediante la classe **Scanner**, e vengono confrontate con il metodo della classe **String** "compareTo".
Infine, si sfrutta un ciclo ***for*** abbreviato sulla lista per stampare le singole parole.

Note:

- Il frammento è estrapolato da un programma generico; non vengono perciò fatti riferimenti alle classi del programma.
- Il metodo che legge le stringhe e crea la lista è statico, perché lo scopo dell'esempio è focalizzarsi sulla realizzazione, non sulla progettazione delle classi. Si ricordi però che l'utilizzo di troppi metodi statici non è considerata una buona pratica di programmazione ad oggetti.
- Il metodo "*sort*" vuole come parametro un **Comparator**, la cui istanziazione non vuol essere oggetto di questa lezione. Non è necessario, al momento, che sappiate cosa sia nello specifico.

Esempio 1.2.2 (continua)

Il programma permette l'inserimento di un numero **arbitrario** di stringhe da linea di comando, finché l'utente non digita il comando "stop".
A quel punto, viene restituita la lista **in ordine** alfabetico.

```
public static void main(String[] args) {
    // Inizio la lettura delle stringhe
    List<String> parole = leggiListaFinoA("stop");
    // Ordino le stringhe utilizzando un Comparator "già pronto":
    // Il metodo "sort" vuole come parametro un oggetto Comparator,
    // che viene creato in automatico dal metodo "compareTo" della classe Stringa.
    parole.sort(String::compareTo);
    // Stampo la lista di parole, che ora è ordinata
    System.out.println("Lista di parole ordinate:");
    // Utilizzo un ciclo for abbreviato
    for (String s : parole) {
        System.out.println(s);
    }
}
```

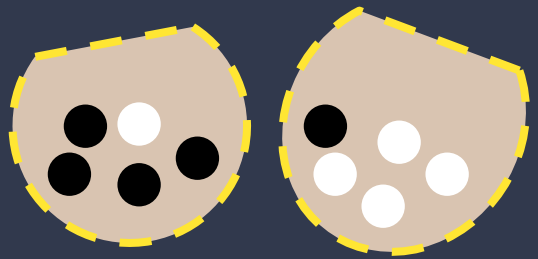
Esempio 1.2.2 (continua)

Dettaglio del metodo “`leggiListaFinoA(stop : String) : List<String>`”

```java
/**
 * Legge una lista di parole finché non si incontra la stringa di stop.
 * Tale stringa viene esclusa dalla lista, e le restanti parole vengono restituite alla fine del metodo.
 */
public static List<String> leggiListaFinoA(String comando_di_stop) {
    // Creazione della lista e dello scanner
    List<String> lista_parole = new LinkedList<String>();
    Scanner scanner = new Scanner(System.in);
    // Lettura della prima parola
    String parola_letta = scanner.nextLine();
    // Lettura di tutte le parole successive, con condizione di terminazione
    while (parola_letta.compareTo(comando_di_stop) != 0) {
        // Aggiungo alla lista la parola letta, che so non essere quella di fine
        lista_parole.add(parola_letta);
        // Leggo una nuova parola
        parola_letta = scanner.nextLine();
    }
    // restituisco la lista alla fine del metodo.
    return lista_parole;
}
```

Esempio 1.2.4

Sacchetti di Biglie
In allegato a queste diapositive, troverete un esempio di progetto quasi-completo che utilizza le liste e buona parte dei metodi visti.
Il programma permette la gestione basilare di alcuni sacchetti di biglie, permette di mischiarli e di estrarre casualmente da essi delle biglie.

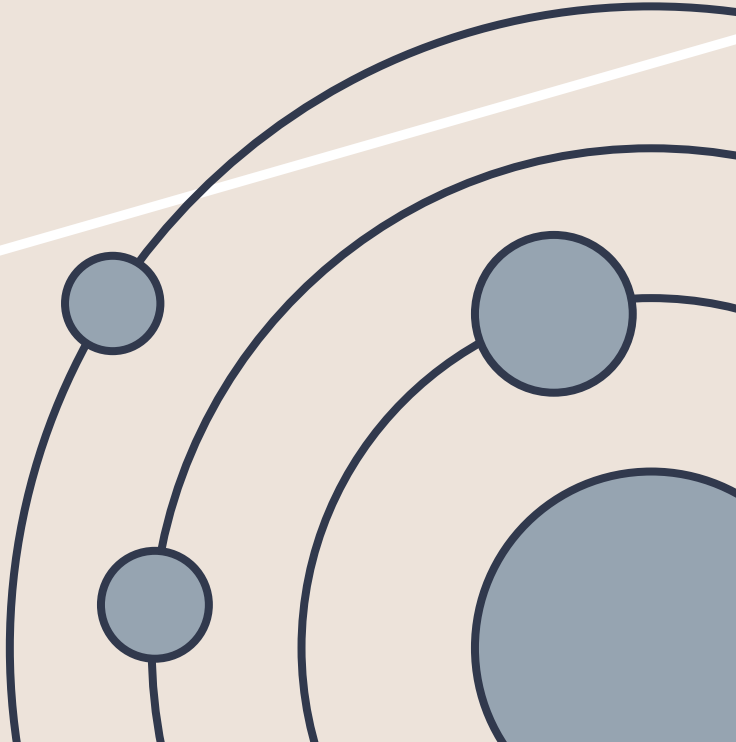


Note:

- Il programma non è per nulla ottimizzato: esistono mille modi per migliorarlo (che vi invito a sperimentare), ma che non sono il focus della lezione.
- La soluzione proposta non è sicuramente l'unica: esistono altre soluzioni (più o meno efficienti) a livello di progettazione che risolvono lo stesso tipo di problema.

Esercizio 1.2.4

Planetarium



Planetarium

Dopo millenni di attesa, il **Consiglio Intergalattico** ha finalmente accettato la richiesta del vostro popolo di entrare a far parte dell'**Impero**, a patto che effettuiate il censimento completo di tutti i pianeti e di tutte le lune facenti parte del vostro sistema stellare.

Come *Team Ufficiale di Programmatori* di tale sistema stellare, i governatori vi chiedono di sviluppare un programma in **Java** (il linguaggio ufficiale dell'Impero) che li aiuti ad organizzare tutte le informazioni in loro possesso. Questo programma dovrà elaborare i dati sulle masse e sulle posizioni di tutti i grossi corpi celesti dell'intero sistema, per poi calcolare il **centro di massa** complessivo.

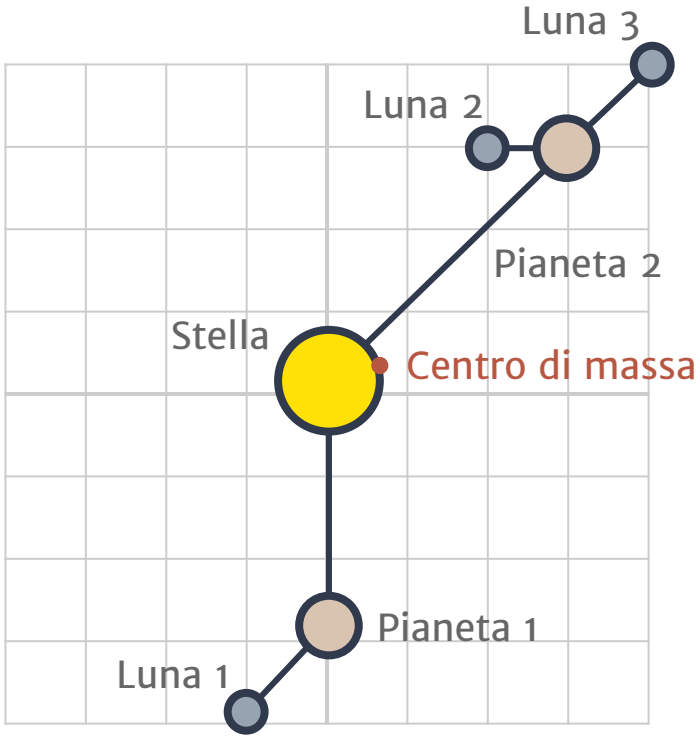
Solo quando invierete la posizione esatta del centro di massa al **Consiglio Intergalattico**, esso potrà iniziare in quel luogo la costruzione dell'ambasciata imperiale; ciò permetterà alle popolazioni che vivono nel campo gravitazionale della vostra Stella di diventare parte dell'Impero, mentre voi Programmatori verrete ricompensati dai governatori dei vostri pianeti con un paio di lune a testa.

Specifiche del programma

Il Consiglio Intergalattico vi richiede di creare un software in Java che **modellizzi il sistema stellare** con tutti i suoi pianeti e le sue lune. Poiché deve essere esportabile (al Consiglio piace riciclare), è necessario che sia sufficientemente "generico" da adattarsi a qualunque sistema stellare.
In particolare, sapete con certezza che:

- Esiste una sola stella nel sistema.
- Ogni pianeta ruota attorno ad una e una sola stella, ed ogni luna ruota attorno ad uno e un solo pianeta.
- Per ogni corpo celeste (stelle, pianeti e lune) si devono salvare le informazioni sulla **massa** e sulla **posizione**. Tale posizione può essere espressa in maniera relativa (quindi rispetto al corpo celeste attorno a cui ruota) o assoluta (secondo un sistema di riferimento arbitrario).
- In caso di nuove scoperte, deve essere possibile aggiungere nuovi pianeti o lune.
- Per distinguere i singoli corpi celesti, è importante identificare ciascuno di essi con un codice univoco.
- Una volta terminato il censimento dei corpi celesti, il programma deve calcolare il centro di massa, facendo la media ponderata sulle masse delle posizioni dei corpi.

Esempio di sistema stellare



Prendiamo, ad esempio, il sistema di ***Cha Halpha 1*** (una stella piccolina, non troppo vicino a noi).

Sole: massa = 30, posizione = (0,0)
Pianeta 1: massa = 5, posizione = (0,-3)
Pianeta 2: massa = 7, posizione = (3,3)
Luna 1: massa = 1, posizione = (-1,-4)
Luna 2: massa = 2, posizione = (2,3)
Luna 3: massa = 1, posizione = (4,4)

Somma delle masse = 46
Somma pesata delle posizioni = (28,12)

Centro di massa = (28/46,12/46)
= (0.608,0.261)

Funzionalità base

Il programma commissionato dovrà soddisfare almeno le seguenti funzionalità base:

- Gestione dei corpi celesti del sistema stellare:
  - Aggiunta di nuovi pianeti o lune, in caso di nuove scoperte.
  - Rimozione di vecchi pianeti o lune, in caso di “catastrofi naturali”.
  - Identificazione di ciascun corpo celeste con un codice univoco.
- Ricerca di un corpo celeste all’interno del sistema:
  - Possibilità di capire se è presente nel sistema stellare.
  - Nel caso di lune, identificazione del pianeta attorno a cui gira.
- Visualizzazione delle informazioni:
  - Dato un pianeta, visualizzazione delle lune che gli orbitano intorno.
  - Data una luna, visualizzazione del percorso [stella > pianeta > luna] necessario per raggiungerla.
- Calcolo del centro di massa su richiesta, sulla base delle informazioni disponibili volta per volta.

Il software deve essere corredato dal rispettivo diagramma UML delle classi.

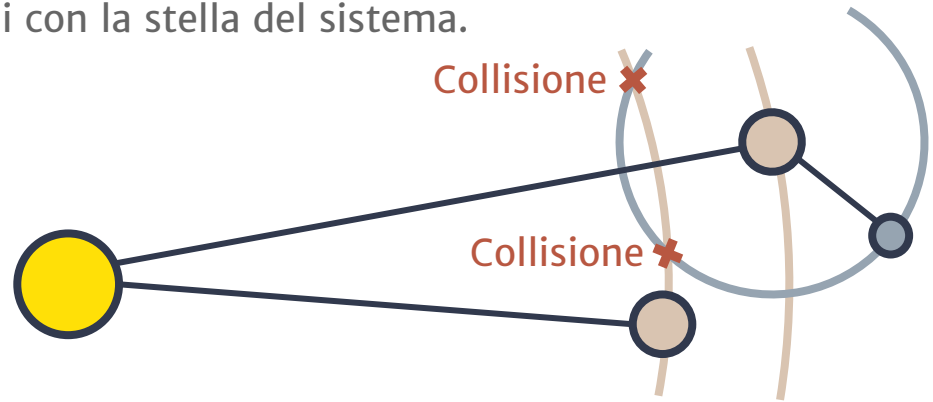
Funzionalità aggiuntive *[Calcolo della rotta]*

In caso la vostra squadra di programmatori finisca particolarmente in fretta, potete proseguire con lo sviluppo del programma implementando le funzionalità riportate qui sotto. Si noti che, sebbene *nessuna di esse vi precluda il pieno raggiungimento della vostra paga*, il Consiglio Intergalattico non potrà non tenerne conto.

- **Calcolo della rotta** fra due corpi celesti:
  - Su richiesta dell'utente, il programma deve mostrare la rotta fra due corpi del sistema stellare selezionati dall'utente stesso.
  - Le leggi imperiali impongono che, nel caso in cui si viaggi fra due corpi di uguale "grado" (due lune, o due pianeti), si faccia **scalo** sul corpo celeste di grado più alto (rispettivamente, il pianeta in comune o la stella del sistema). In questo modo, la rotta fra due corpi celesti qualunque è **unica**, e l'Impero riesce a mantenere l'ordine.
  - La rotta deve essere rappresentata come sequenza di corpi celesti (ad esempio: Luna 1 > Pianeta 1 > Stella > Pianeta 2).
  - Deve essere indicata la distanza totale che si percorrerebbe seguendo la sequenza sopra indicata (nell'esempio precedente: 8,65).

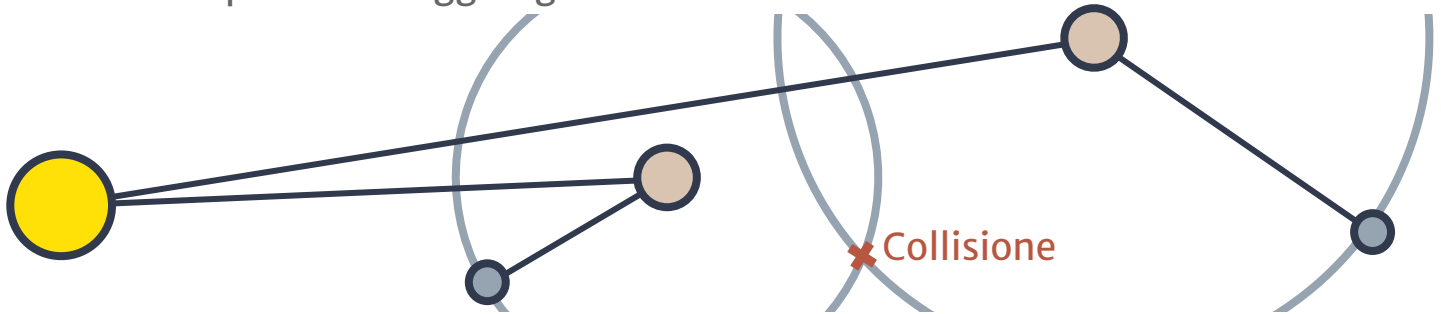
Funzionalità aggiuntive *[Collisioni - 1]*

- **Calcolo della collisione** fra i corpi celesti:
  - Su richiesta dell'utente, il programma deve stabilire se sia possibile che due corpi qualunque del sistema collidano uno contro l'altro. Non è necessario stabilire quando e dove, né quali corpi siano, ma solamente **se** questo possa succedere.
  - Due corpi possono collidere se e solo se esiste una configurazione del sistema per cui due corpi possono trovarsi nella stessa posizione puntuale.
  - Ogni corpo ha distanza fissa dal corpo attorno a cui ruota (detta *raggio di rivoluzione*). Pertanto, nessuna luna collliderà mai col proprio pianeta e nessun pianeta collliderà mai con la stella del sistema.

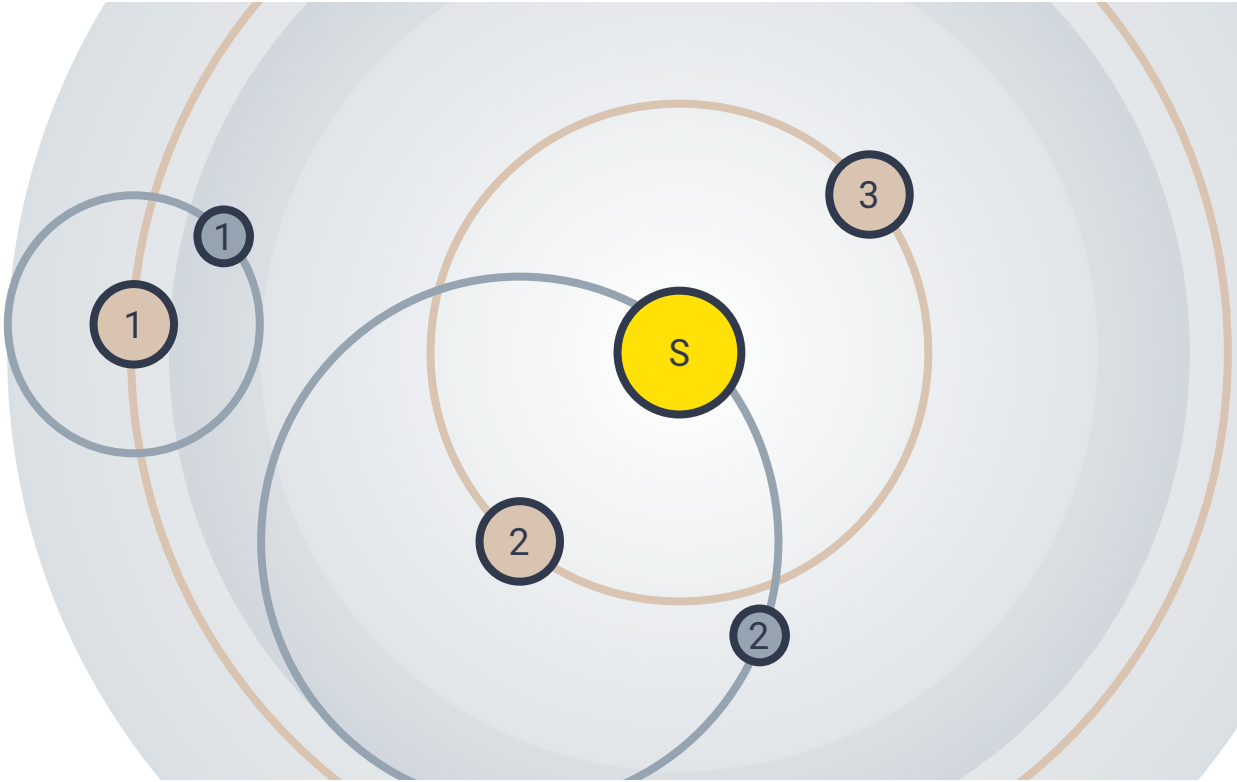


Funzionalità aggiuntive *[Collisioni - 2]*

- ...
- In particolare, se ne può dedurre che:
  - Due pianeti potranno collidere fra di loro se e solo se hanno lo stesso raggio di rivoluzione.
  - Due lune orbitanti attorno allo **stesso** pianeta potranno collidere se e solo se hanno lo stesso raggio di rivoluzione.
  - Due lune orbitanti attorno a pianeti **differenti** potranno collidere se e solo se la corona circolare (con centro la stella) che rappresenta tutte le posizioni raggiungibili della prima luna interseca la corona circolare (con centro la stella) di tutte le posizioni raggiungibili della seconda luna.



Funzionalità aggiuntive *[Collisioni - esempio]*



Elenco di tutte le collisioni possibili nel sistema a sinistra, composto da 6 corpi celesti:

	S	1	2	3	1
2	✖			✖	✖
1					
3			✖		
2					
1					

Assunzioni

Per facilitare il vostro compito, potete assumere che:

- Tutti i corpi celesti e le loro orbite appartengano allo **stesso piano**; le coordinate, pertanto, possono essere rappresentate con solo due valori.
- La posizione di ciascun corpo corrisponda alla posizione del suo centro. Per il calcolo delle posizioni o delle eventuali distanze, i corpi possono essere considerati **puntiformi**.
- Ogni orbita è **circolare**.
- Se un corpo A può collidere con un corpo B, allora il corpo B può collidere con il corpo A (ossia: la relazione di collisione è **simmetrica**).
- Poiché bisogna solo censire i corpi celesti, le informazioni che avrete su di essi saranno "immutabili" (in particolare, faranno riferimento alla posizione prevista per la mezzanotte del 31 dicembre 2156). Non c'è bisogno di calcolare gli spostamenti, e il calcolo del centro di massa dovrà essere fatto sulle informazioni costanti.
- Per limiti fisici, non esistano mai più di:
  - 26.000 pianeti attorno ad una stella.
  - 5.000 lune attorno ad un pianeta.
- Il programma preveda l'interazione con l'utente solo tramite **linea di comando** (un'interfaccia grafica è considerata una grave offesa sul pianeta Magrathea).

Modalità di consegna

Una volta terminato il programma, esso dovrà essere inviato dal responsabile del gruppo per email a pgmarnaldo@googlegroups.com, con oggetto "*Programma Arnaldo 2019 – Esercizio 1.2.4*" entro la scadenza sotto riportata. Il codice può essere inviato attraverso qualunque mezzo ritenuto adatto e comodo allo scopo.

Nella mail, è richiesto di specificare il nome del gruppo e degli altri componenti.

Il lavoro consegnato contribuirà alla valutazione complessiva degli studenti partecipanti.

Scadenza di consegna:
ore 23:59, venerdì 1 Maggio 2020

Bibliografia e risorse online

- Documentazione dell'interfaccia List: https://docs.oracle.com/javase/8/docs/api/java/util/List.html
- Spiegazione dell'allocazione dinamica di memoria all'interno di *ArrayList*: https://www.quora.com/How-does-Arraylist-in-java-internally-implemented-Is-it-change-its-size-to-double-Then-Why-double
- **Esempio di lista concatenata in Java** (diverso dalla *LinkedList*, ma permette di capire come funziona) http://www.simplesoft.it/una-lista-concatenata-in-java.html
- Confronto fra *ArrayList* e *LinkedList* https://manifesto.co.uk/java-lists-arraylist-vs-linkedlist/
- **Iterator:** https://www.geeksforgeeks.org/how-to-use-iterator-in-java/

Questa presentazione è stata realizzata da
Michele Dusi
ed è stata aggiornata da
Ilaria Pasini
per lo Student Branch IEEE
dell'Università degli Studi di Brescia,
in occasione del Programma Arnaldo 2019.

UNIVERSITÀ DEGLI STUDI DI BRESCIA

IEEE