UNIVERSITÀ DEGLI STUDI DI BERGAMO
Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

Implementazione di codice algoritmico in Java

PROGETTAZIONE, ALGORITMI E COMPUTABILITÀ (38090-MOD1)

Corso di laurea Magistrale in Ingegneria Informatica

RELATORE
Prof.ssa Patrizia Scandurra

SEDE
DIGIP

Argomenti dell'esercitazione

- Implementare un algoritmo in Java
  - Interfacce, ereditarietà e polimorfismo
  - Tipi generici
  - Test di uguaglianza
  - Ordinamento naturale
  - Sequenze
    (del *Java Collection Framework* **`java.util`**)
  - Iteratori, stream

Interfaccia

- Stabilisce la struttura di un oggetto/componente o di astrazione di dato (*Abstract Data Type - ADT*), ma non un'implementazione!
- Un'interfaccia:
  - contiene **i prototipi dei metodi**, ma non i corpi degli stessi
  - può contenere **attributi**, ma queste sono implicitamente ***static* e *final***
  - può essere dichiarata **public** (solo se definita in un file con lo stesso nome) o con visibilità di package

```
interface Instrument {
 // Compile-time constant:
 int i = 5; // static & final
 // Cannot have method definitions:
  void play();
  String what();
  void adjust();
}
```

Interfacce, ereditarietà e polimorfismo

- Java fornisce solo ***ereditarietà singola***
- In Java **una classe può** però **ereditare da più interfacce --** Java supporta l'**ereditarietà multipla** tramite interfacce!
- Sintassi:

```
interface A { … }
interface B { … }
interface C { … }
class MyClass implements A, B, C { … }
```

- Si può ereditare da quante interfacce si vuole, ciascuna è un tipo indipendente verso il quale si può effettuare l'***upcasting (conversione larga)***: conversione *safe* da sottotipo a un supertipo

```
MyClass o = new MyClass();
A ao = (A) o; //upcasting
B bo = (B) o; //upcasting
```

```
A ao = new MyClass(); //upcasting
B bo = new MyClass(); //upcasting
```

Ereditarietà tra interfacce

Usando l'ereditarietà tra interfacce è possibile:

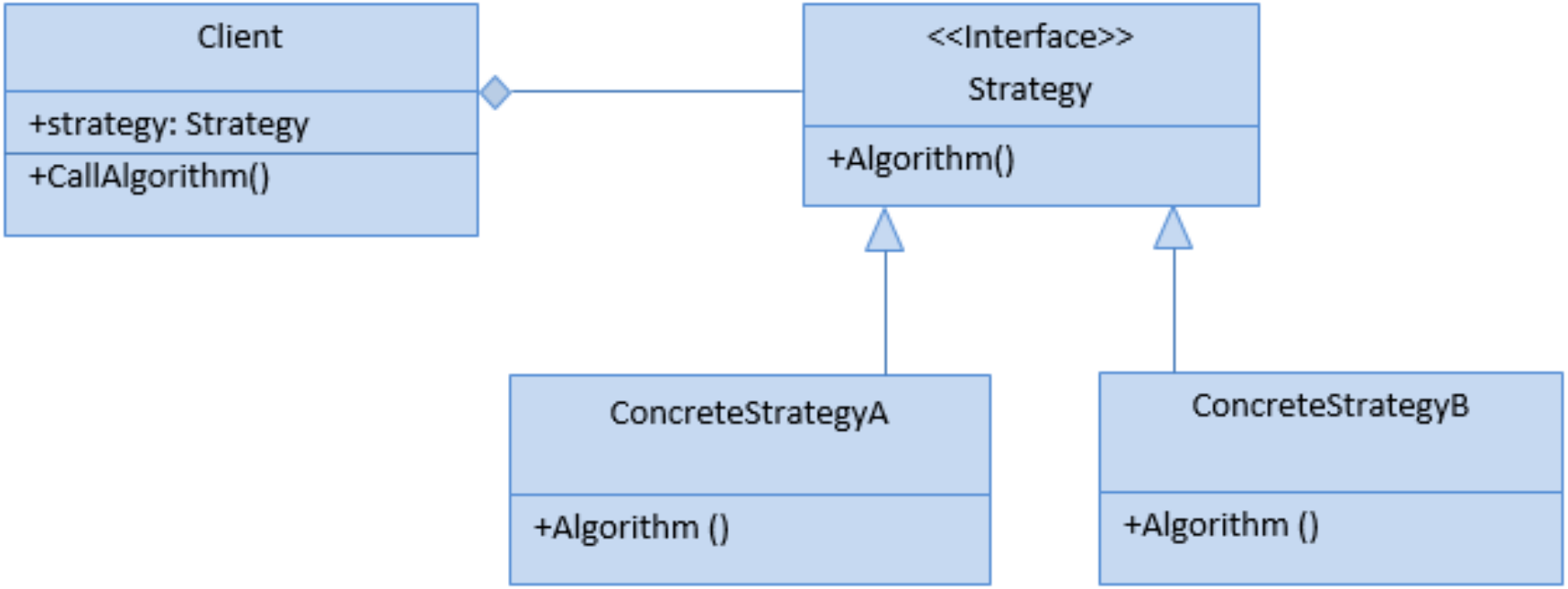
- aggiungere nuovi metodi alle interfacce
- combinare diverse interfacce fra loro in una nuova interfaccia

```
interface Monster { void menace(); }

interface DangerousMonster extends Monster {
    void destroy(); }

interface Lethal { void kill(); }

class DragonZilla implements DangerousMonster {
    public void menace() {...}
     public void destroy() {…} }

interface Vampire extends DangerousMonster,
    Lethal {
     void drinkBlood(); }

public class HorrorShow {
    static void u(Monster b) { b.menace(); }
    static void v(DangerousMonster d) {
       d.menace();
       d.destroy(); }
    public static void main(String[] args) {
      Monster if2 = new DragonZilla();
       u(if2);
       v(if2); }
}
```

Stessa interfaccia, diverse implementazioni

Ci aspettiamo che implementazioni algoritmi diversi per uno stesso problema computazionale condividano la medesima interfaccia.

*Soluzione: Design pattern **Strategy*** - le classi che implementano l'interfaccia rappresentano le *varianti* dell'algoritmo.



Stessa interfaccia, diverse implementazioni

- **Definisci prima l'interfaccia**:

```
public interface AlgoDup {
     public boolean verificaDup(List S);
}
```

- **Per ogni possibile implementazione** dell'algoritmo, **definisci una classe** che **implementa l'interfaccia**:

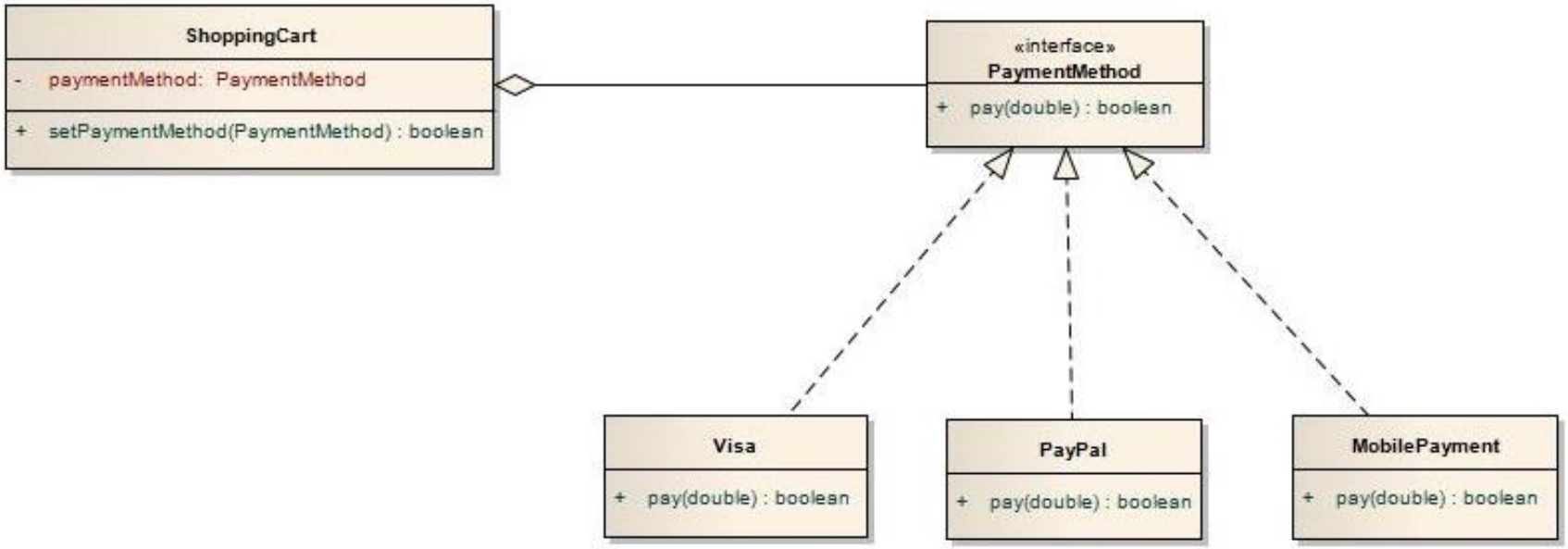
```
public class verificaDupList implements AlgoDup {
 public boolean verificaDup(List S){…}
}
public class verificaDupOrdList implements AlgoDup {
 public boolean verificaDup(List S){…}
}
```

- **Dichiara poi un oggetto usando come tipo l'interfaccia** -- la scelta dell'algoritmo sarà così delegata all'operatore *new (upcasting)*

```
AlgoDup myAlg = new VerificaDupOrdList();
boolean result = myAlg.verificaDup(S);
```

Stessa interfaccia, diverse implementazioni

- **Perché usiamo il pattern Strategy**?
  - **se cambia l'implementazione**, **non cambia l'interfaccia** e quindi **non cambiano le dipendenze della componente** con il resto del programma (codice client)
  - consente di **cambiare dinamicamente quale implementazione dell'algoritmo utilizzare** a seconda delle diverse esigenze



Tipi generici e boxing

Java consente di definire tipi generici:

- Tramite la **classe Object**
  - Tipi primitivi inscatolati nelle **Classi wrapper** (Integer,Float, ..)
    - Gestione agevolata da Java 5 con l'**auto-boxing**:
      - Ad esempio, data una lista `List S = new LinkedList();` possiamo scrivere direttamente `S.add(5)` invece di `S.add(new Integer(5))` per aggiungere un elemento ad S
- Tramite i **tipi generici** (da Java 5):

  ```
  List<Integer> myIntList = new LinkedList<Integer>();
  ```

  equivale a livello di codice a dichiarare:

  ```
  List myIntList = new LinkedList(); //Lista di Object
  ```

  e ad eseguire implicitamente le conversioni Object->Integer e Integer->Object per leggere e scrivere gli elementi

Recap sui tipi generici in Java

- **List<E>** è un **tipo generico** (ma anche una **interfaccia generica**)
- **ArrayList<E>** è una **classe generica** che **implements** List<E>

E è **type variable/ type parameter**

```
public class ArrayList<E>
implements List<E> {
  private E[] elementData;
  private int size;
  ... //stuff
  public boolean add(E o) {
    elementData[size++] = o;
    return true;
  }
  public E get(int i) {
    return elementData[i];
  } //etc.
}
```

E "diventa" Money

```
List<Money> S = new ArrayList<Money>();
```

List/ArrayList entrambe **parametrizzate** con Money

```
public class ArrayList
implements List<Money> {
  private Money[] elementData;
  private int size;
  ... //stuff
  public boolean add(Money o) {
    elementData[size++] = o;
    return true;
  }
  public Money get(int i) {
    return elementData[i];
  } //etc...
}
```

Test di uguaglianza di oggetti

- Vogliamo il test dei “valori” non dei riferimenti!
  - **NON usare: equality (==) inequality (!=)**

- In Java, sfruttiamo il polimorfismo del **metodo equals()** ereditato dalla classe Object:

```
if (name.equals("Mickey Mouse")) …
```

  - **va ridefinito in ogni (user) classe** (altrimenti, il comportamento di default è lo stesso di ==)
  - deve implementare una **relazione di equivalenza** tra riferimenti (non nulli) ad oggetti

Test di uguaglianza di oggetti

```
public class Person {
 String title;
 String fullName;
 int age;
 public Person(String title, String fullName, int age){
                  this.title = title;
                  this.fullName = fullName;
                  this.age = age;
        }
 //Metodi accessori
 String getFullName() {  return fullName; }
   ...
}
```

Test di uguaglianza di oggetti

- ATTENZIONE: **L'argomento del metodo equals deve essere Object** (altrimenti sarebbe un overloading!)

```
public class Person {
    …
    public boolean equals(Object obj) {
        if(this == obj) { return true;}
        if (!(obj instanceof Person)) { return false;}
        Person person = (Person)obj; //Cast
        return age == person.getAge() &&
                fullName.equals(person.getFullName())
                && title.equals(person.getTitle());
         }
    …
    }
```

Ordinamento naturale con interfaccia Comparable<T>

- **Confronto di tipi primitivi:** gli **operatori relazionali** < <= > >= ==
- **Confronto di oggetti :** la classe dell'oggetto deve implementare l'interfaccia generica Comparable<T> o Comparable (non generica) e implementare il metodo **compareTo**

```
public interface Comparable<T>{
        int compareTo(T o);
    }
//Uso con a e b oggetti dello stesso tipo:
a.compareTo(b);//return < 0, > 0, == 0
```

Ordinamento naturale: esempio

Definire **compareTo dell'interfaccia Comparable<T>**

- Non servono cast
- Si noti l'uso "ricorsivo" di compareTo

```
class Person implements Comparable <Person> { ...
public int compareTo(Person another) {
 if (this.fullname.compareTo(another.getFullname())<0)
   return -1:
 if (this.fullname.compareTo(another.getFullname())>0)
   return 1;
 return this.age - another.getAge();
  }
}
```

Ordinamento naturale: esempio

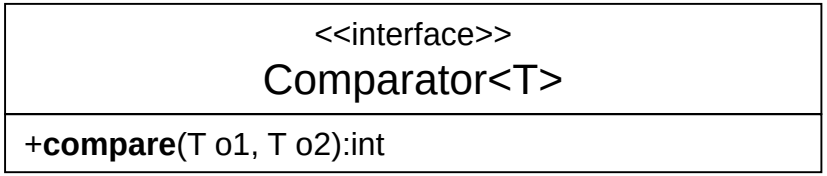
Definire il **compareTo dell'interfaccia Comparable**

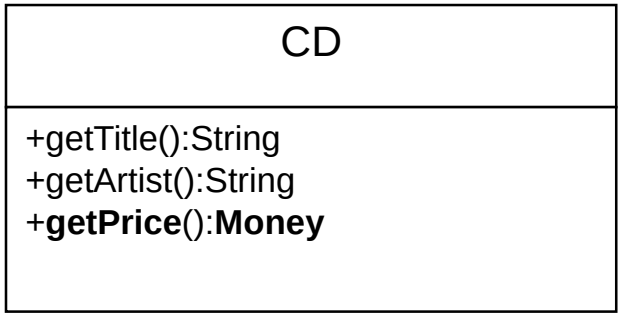
- Occorre un cast!

```
class Person implements Comparable { ...
public int compareTo(Object another) throws
   ClassCastException {
 if (!(another instanceof Person))throw new
   ClassCastException("A Person object expected.");
 Person anotherP = (Person) another;  //cast
 if (this.fullname.compareTo(anotherP.getFullname())<0)
   return -1:
 if (this.fullname.compareTo(anotherP.getFullname())>0)
   return 1;
 return this.age - anotherP.getAge();
   }
}
```

Ordinamento naturale con interfaccia Comparator

- Ulteriore metodo: definire un oggetto che implementa l'interfaccia **java.util.Comparator<T>** per confrontare elementi di una certa natura






```
public class PriceComparator
implements Comparator<CD> {
  public int compare(CD c1, CD c2) {
    return c1.getPrice().compareTo(c2.getPrice());
  }
}
```

Gestione degli errori

- Un algoritmo tipicamente assume che i dati in ingresso rispettino certe condizioni (*precondizioni*)
  - Ad es. un metodo che calcola la radice quadrata di un numero, assume che tale numero sia >=0
- Se ciò non avviene, l'esecuzione del metodo non può essere (e non deve essere!) avviata e/o portata a termine
- **Java offre:**
  - **Il meccanismo delle eccezioni** per alterare il normale flusso del controllo di un programma
  - L'**istruzione *assert espr;*** permette di verificare se una data espressione è vera o falsa; se falsa viene sollevata un'eccezione di tipo *AssertionError*
    - Utile per implementare invarianti*: pre-condizioni, post-condizioni, invarianti interne, invarianti di classe*
    - Di default, sono disabilitate; vanno abilitate con il comando java che avvia l'applicazione

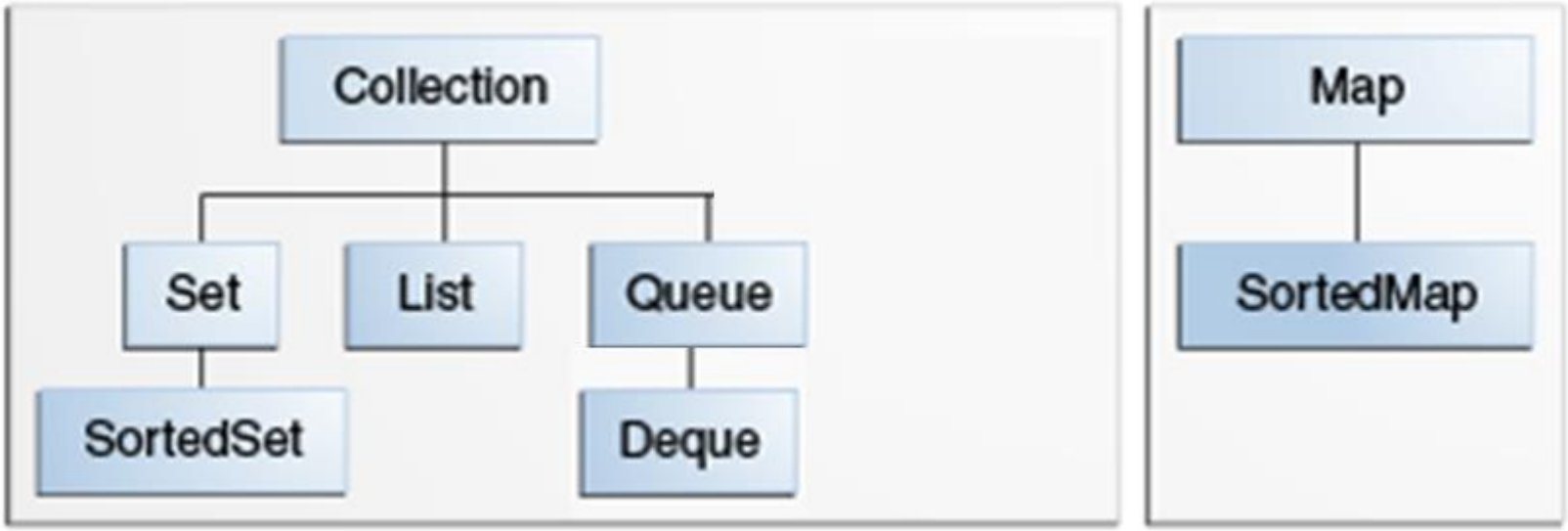
Esercizio 1

- Definire una user-classe **Studente** che ridefinisce i metodi equals() e compareTo()

Il Java Collection Framework (JCF)

- **Interfacce e classi del package java.util**
- Forniscono **ADT** e **strutture dati** per manipolare **collezioni di oggetti**
- E **algoritmi di base** (ad es. come ordinamento e ricerca)
- http://docs.oracle.com/javase/tutorial/collections/index.html

JCF – le interfacce



- Formano una gerarchia
- Tutte le **interfacce** sono **generiche**. Ad esempio la definizione dell'interfaccia Collection:

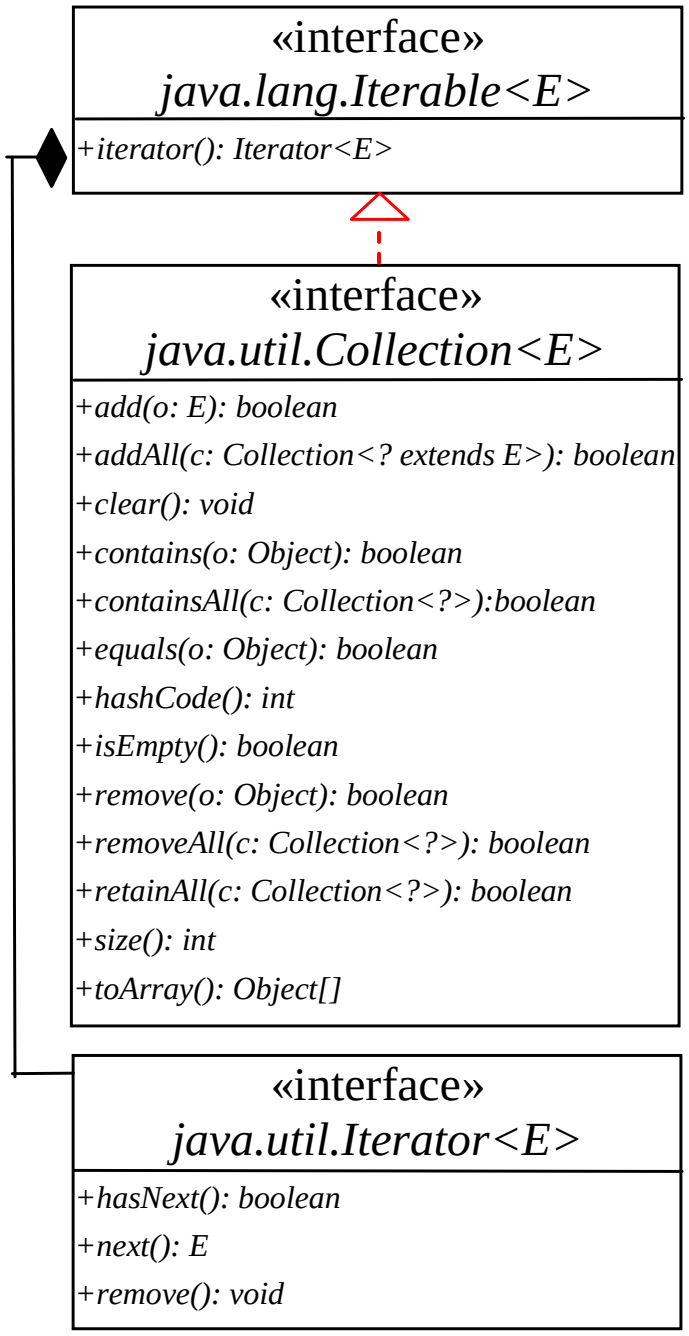
  public interface Collection<E>...

L’interfaccia e classi predefinite per le sequenze

- **Collection**: nessuna ipotesi sul tipo di collezione; no implementazioni dirette
- **List**: introduce l’idea di sequenza (collezione ordinata, possibili duplicati)

- Classi per l’interfaccia **`List`**: **`ArrayList`**, **`LinkedList`**, **`Vector`**, **`Stack`**

Interfaccia Collection



Returns an iterator for the elements in this collection.

Adds a new element o to this collection.

Adds all the elements in the collection c to this collection.

Removes all the elements from this collection.

Returns true if this collection contains the element o.

Returns true if this collection contains all the elements in c.

Returns true if this collection is equal to another collection o.

Returns the hash code for this collection.

Returns true if this collection contains no elements.

Removes the element o from this collection.

Removes all the elements in c from this collection.

Retains the elements that are both in c and in this collection.

Returns the number of elements in this collection.

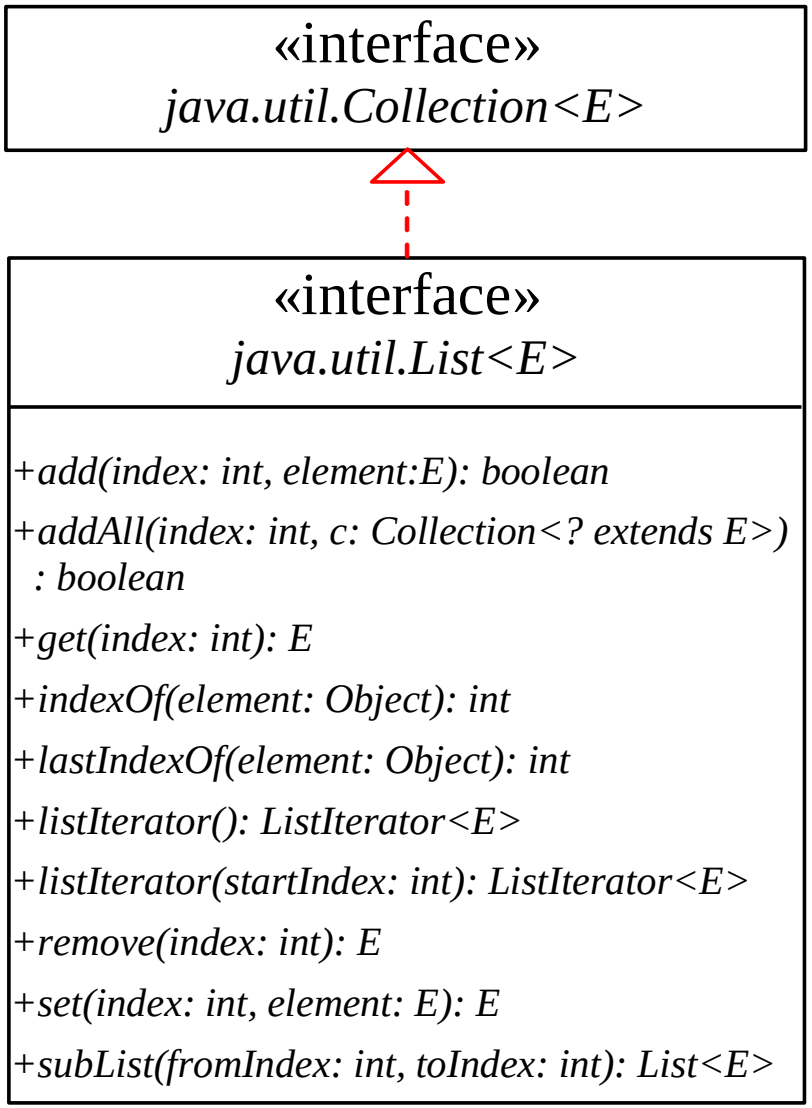
Returns an array of Object for the elements in this collection.

Returns true if this iterator has more elements to traverse.

Returns the next element from this iterator.

Removes the last element obtained using the next method.

Interfaccia List

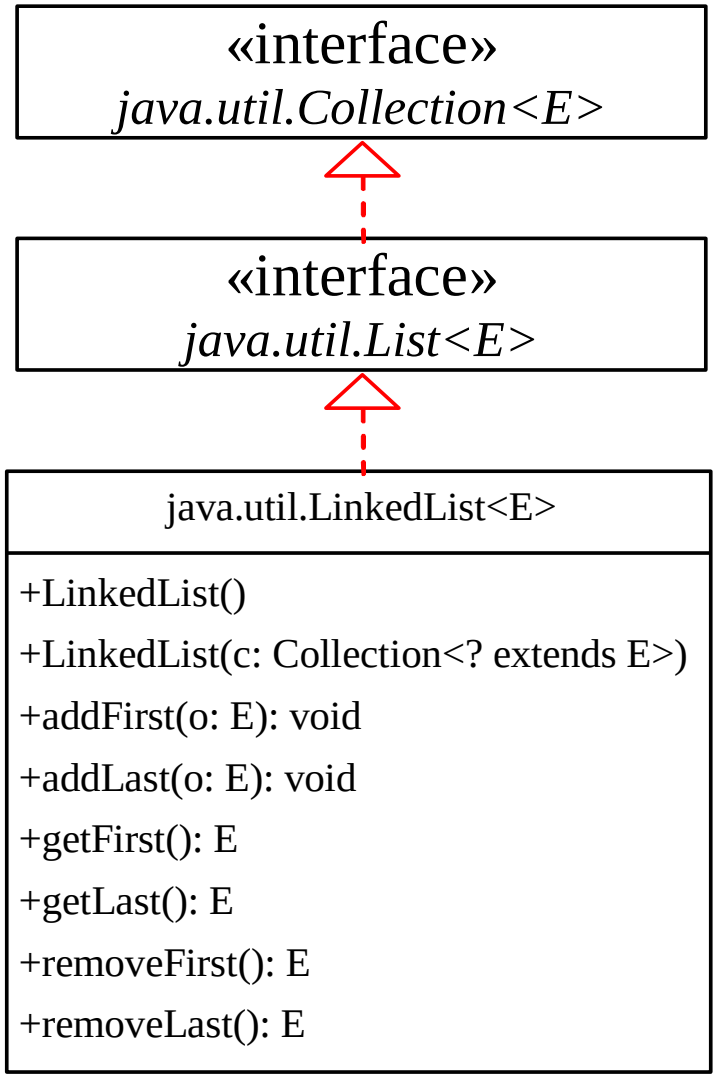


Sono presenti i metodi **`get/set`** per l'**accesso posizionale**!

+add(index: int, element:E): boolean	Adds a new element at the specified index.
+addAll(index: int, c: Collection<? extends E>) : boolean	Adds all the elements in c to this list at the specified index.
+get(index: int): E	Returns the element in this list at the specified index.
+indexOf(element: Object): int	Returns the index of the first matching element.
+lastIndexOf(element: Object): int	Returns the index of the last matching element.
+listIterator(): ListIterator<E>	Returns the list iterator for the elements in this list.
+listIterator(startIndex: int): ListIterator<E>	Returns the iterator for the elements from startIndex.
+remove(index: int): E	Removes the element at the specified index.
+set(index: int, element: E): E	Sets the element at the specified index.
+subList(fromIndex: int, toIndex: int): List<E>	Returns a sublist from fromIndex to toIndex.

La classe `LinkedList`

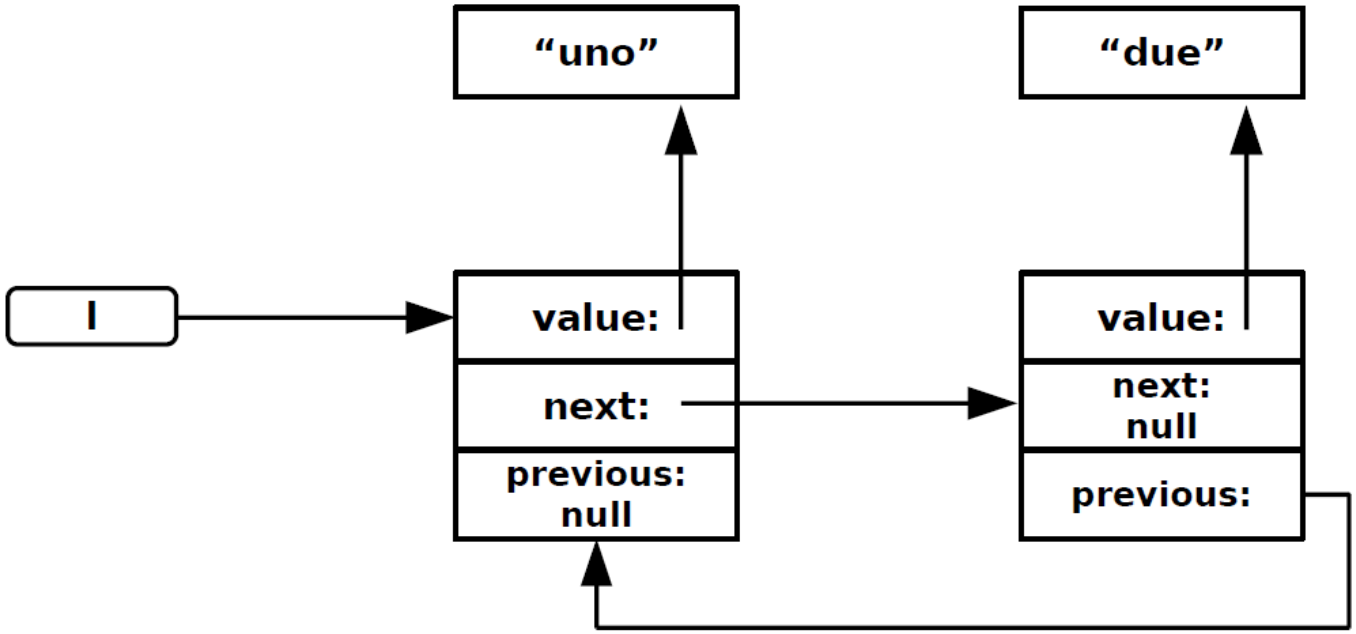
- Rappresenta una lista **doppiamente concatenata**



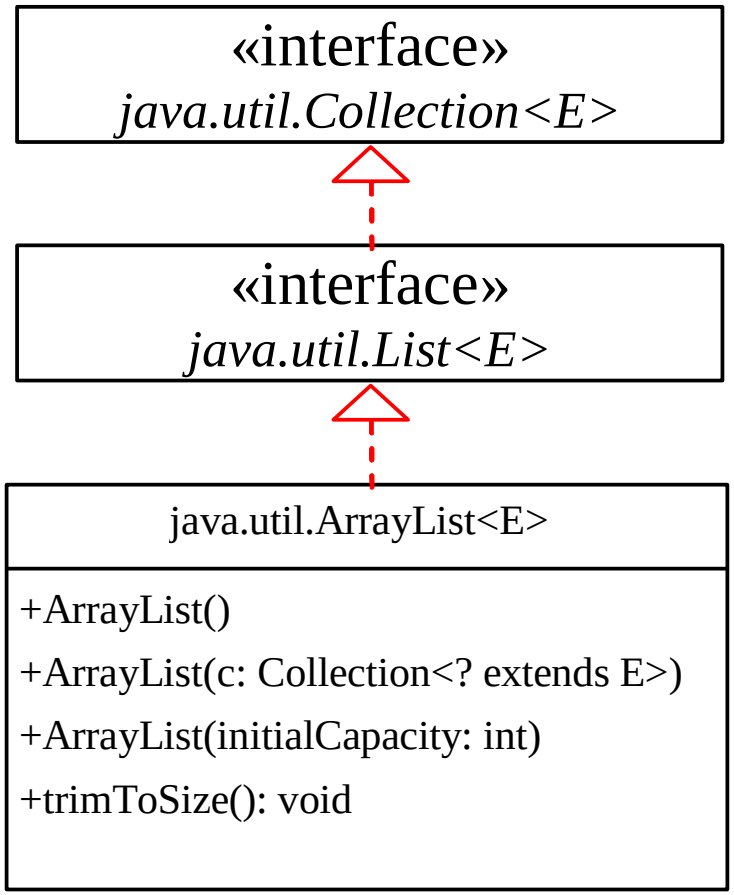
Esempio LinkedList

```
LinkedList<String> l = new LinkedList<String>();

l.add(“uno”);
l.add(“due”);
```

Memory layout:



La classe **`ArrayList`**



Creates an empty list with the default initial capacity.

Creates an array list from an existing collection.

Creates an empty list with the specified initial capacity.

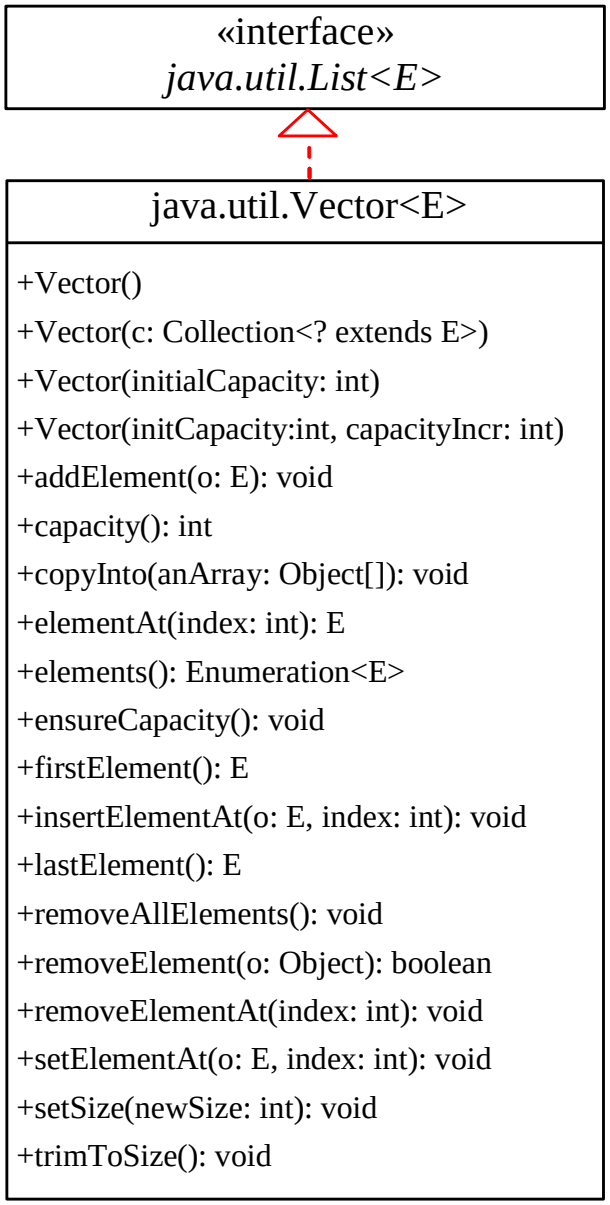
Trims the capacity of this ArrayList instance to be the list's current size.

- ArrayList realizza List con un **array di dimensione dinamica**
- Il ridimensionamento avviene in modo che l'operazione di inserimento (add) abbia complessità *ammortizzata* costante

Le liste e l'accesso posizionale

- In LinkedList, ciascuna operazione di accesso posizionale può richiedere un tempo proporzionale alla lunghezza della lista
  - per accedere all'elemento di posto i è necessario scorrere la lista, a partire dalla testa o dalla coda, fino a raggiungere la posizione desiderata in O(i) passi
- In ArrayList, ogni operazione di accesso posizionale richiede tempo costante – O(1) passi
- Pertanto, è **fortemente sconsigliato utilizzare l'accesso posizionale su LinkedList**

La classe `Vector`



- Simile ad ArrayList ma Vector contiene la versione synchronized dei metodi per accedere e modificare l'array

+Vector()	Creates a default empty vector with initial capacity 10.
+Vector(c: Collection<? extends E>)	Creates a vector from an existing collection.
+Vector(initialCapacity: int)	Creates a vector with the specified initial capacity.
+Vector(initCapacity:int, capacityIncr: int)	Creates a vector with the specified initial capacity and increment.
+addElement(o: E): void	Appends the element to the end of this vector.
+capacity(): int	Returns the current capacity of this vector.
+copyInto(anArray: Object[]): void	Copies the elements in this vector to the array.
+elementAt(index: int): E	Returns the object at the specified index.
+elements(): Enumeration<E>	Returns an enumeration of this vector.
+ensureCapacity(): void	Increases the capacity of this vector.
+firstElement(): E	Returns the first element in this vector.
+insertElementAt(o: E, index: int): void	Inserts o to this vector at the specified index.
+lastElement(): E	Returns the last element in this vector.
+removeAllElements(): void	Removes all the elements in this vector.
+removeElement(o: Object): boolean	Removes the first matching element in this vector.
+removeElementAt(index: int): void	Removes the element at the specified index.
+setElementAt(o: E, index: int): void	Sets a new element at the specified index.
+setSize(newSize: int): void	Sets a new size in this vector.
+trimToSize(): void	Trims the capacity of this vector to its size.

Iterare una collezione

Esistono 3 modi:

1. **Il costrutto foreach**
2. **Iteratori**
3. **Streaming API** (da java 8)

Il ciclo for-each

- In generale, **il ciclo for-each funziona su tutti gli oggetti che implementano l'interfaccia** Iterable<E>
- Il ciclo **for-each funziona** sugli **array e** anche sulle **collezioni**

```
String[] array = {“uno”, “due”, “tre”};
for (String s: array)
   System.out.println(s);
```

```
for (Object o : collection)
     System.out.println(o);
```

```
for(Object x : coll){ /* operazioni su x */ }
```

equivale a:

```
for (Iterator i =coll.iterator(); i.hasNext(); )
   {/* operazioni su x = i.next() */}
```

Iteratori

- **Oggetti** (delle API JCF) **associati ad una collezione** che **permettono** di effettuare **l'operazione di visita** degli elementi della collezione **in modo efficiente**
- Per ottenere un iteratore per una data collezione, si invoca su essa l'apposito metodo iterator()
- Ogni iteratore offre:
  - un metodo next() che restituisce "il prossimo" elemento
  - un metodo hasNext() per sapere se ci sono altri elementi

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();    // operazione opzionale
}
```

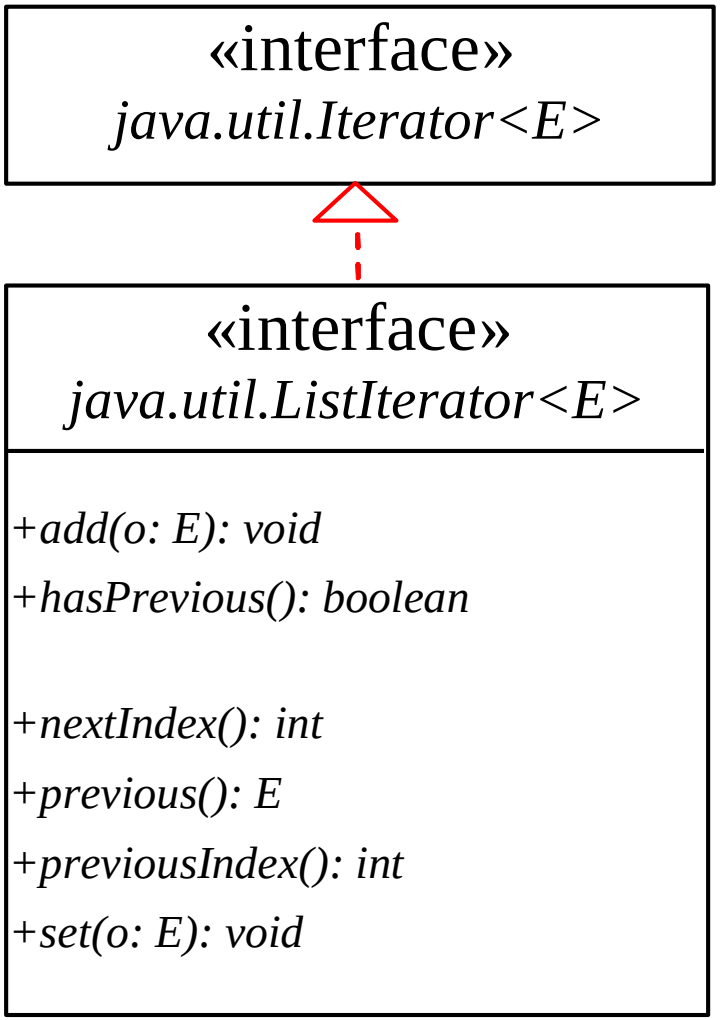
Iteratori: esempio

```
Iterator i = S.iterator(); //S è una qualunque
                           //collezione delle API JCF
while(i.hasNext()){ //se c'è ancora un elemento…
  //preleva l'elemento con i.next()
  Object e = i.next();
  //e lo utilizza
  System.out.println(e);
}
```

- Per usare un **iteratore generico <T>** (next() ha tipo T)

```
Iterator<String> i = S.iterator();
String e = i.next();
```

ListIterator



`netIndex()/previousIndex()`

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/List.html

Adds the specified object to the list.

Returns true if this list iterator has more elements when traversing backward.

Returns the index of the next element.

Returns the previous element in this list iterator.

Returns the index of the previous element.

Replaces the last element returned by the previous or next method with the specified element.

Streaming API

- Usano **IOC** (***Inversion Of Control***): è il framework a controllare l'iterazione e non il programmatore
- L'interfaccia *Stream* è definita nel package *java.util.stream*
- Ampio uso di *espressioni lambda* e di funzioni di aggregazione definite su stream

```
myShapesCollection.stream()
.filter(e -> e.getColor() == Color.RED)
.forEach(e -> System.out.println(e.getName()));
```

```
myShapesCollection.parallelStream()
.filter(e -> e.getColor() == Color.RED)
.forEach(e ->
System.out.println(e.getName()));
```

Multi-core architecture!

Esercizio 2: scrittura di codice algoritmico

- Fornire un'implementazione dell'algoritmo **verificaDup** per il problema dei *Duplicati*
  - Scegliere se implementare l'iterazione usando oggetti Iterator o non
- Testare l'algoritmo su una collezione S di oggetti "studenti"

```
algoritmo verificaDup(sequenza S)
    for each elemento x della sequenza S do
        for each elemento y che segue x nella sequenza S do
            if x = y then return true
    return false
```

Esercizio 3

- Si consideri il tipo di dato astratto Pila

tipo `Pila`:
dati:
una sequenza S di n elementi.

operazioni:
`isEmpty`$() \rightarrow result$
restituisce `true` se S è vuota, e `false` altrimenti.

`push`$(elem\ e)$
aggiunge e come ultimo elemento di S.

`pop`$() \rightarrow elem$
toglie da S l'ultimo elemento e lo restituisce.

`top`$() \rightarrow elem$
restituisce l'ultimo elemento di S (senza toglierlo da S).

- Implementare il tipo di dato astratto Pila sfruttando una struttura dati di tipo *List* del JCF
  - Due possibili modi:
    1. **Per composizione**: la lista di elementi è un attributo della classe che implementa la Pila
    2. **Per ereditarietà**: la classe che implementa la Pila estende una classe del JCF che implementa List (LinkedList, ArrayList, o Vector)