Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Università degli Studi dell'Aquila

Corso di Algoritmi e Strutture Dati con Laboratorio

Java Collections Framework (I parte)

Java Collections Framework

- L'infrastruttura JCF è una raccolta di interfacce e classi, tra loro correlate, appartenenti al pacchetto `java.util`.
- Un esemplare di una classe di tale infrastruttura rappresenta generalmente una raccolta (collection) composta da elementi.
- Queste classi possono usare **tipi parametrici**, in modo che un utente possa specificare il tipo a cui appartengono gli elementi della raccolta nel momento in cui dichiara un esemplare di tale raccolta

Raccolte

- Una raccolta (collection) è un oggetto composto da elementi.
- Esempio: un array è una raccolta di elementi dello stesso tipo che vengono memorizzati in aree contigue della memoria

```
String[] names = new String[5];
```

Array

- Vantaggio:
  - Si può accedere ad un singolo elemento dell'array in modo diretto (proprietà di accesso casuale)
- Svantaggi:
  - La dimensione è fissa. Se la dimensione si rivela insufficiente occorre creare un array più grande e copiarvi il contenuto di quello più piccolo.
  - L'inserimento o la rimozione di un elemento può richiedere lo spostamento di molti elementi.
  - Queste operazioni di mantenimento devono essere gestite dal programmatore

“Collection class”

- Un’alternativa all’uso degli array ?
  L’uso di esemplari di classi che rappresentano raccolte (con abuso di terminologia “*collection class*”)
- Una collection class è una classe i cui singoli esemplari sono raccolte di elementi
- Gli elementi in un’istanza di una collezione devono essere riferimenti ad un oggetto.
  - Non possiamo creare un esemplare di una raccolta i cui singoli elementi siano di tipo primitivo
  - Possiamo usare le classi wrapper

Strutture di memorizzazione

- Strutture di memorizzazione per classi che rappresentano raccolte:

1. **Contiguous collection** (Raccolta contigua): Il modo più semplice per archiviare in memoria una raccolta prevede di memorizzare in un array i riferimenti ai singoli elementi: in pratica la classe ha un array come campo

Strutture di memorizzazione

2. **Linked collection**: invece di usare la contiguità, gli elementi possono essere correlati tra loro mediante collegamenti (link), ovvero riferimenti.
   - In una classe che realizza una raccolta mediante collegamenti, ciascun elemento presente in un suo esemplare è memorizzato in una entry o nodo, che contiene almeno un collegamento ad un altro nodo.

Linked collection

- **Singly-linked list** (lista semplicemente concatenata): raccolta realizzata mediante collegamenti, dove ciascun nodo contiene un elemento ed un riferimento al nodo successivo presente nella raccolta
- **Doubly-linked list** (lista doppiamente collegata): raccolta realizzata mediante collegamenti, dove ciascun nodo contiene un elemento, un riferimento al nodo precedente ed un riferimento al nodo successivo presente nella raccolta
- **Binary search tree**…

Tipi parametrici

- La versione 1.5 ha introdotto una nuova, importante funzionalità nel linguaggio: la programmazione parametrica, in Inglese anche detta *generics*.
- Si tratta della possibilità di **specificare il tipo di un elemento dotando classi, interfacce e metodi di parametri di tipo.** Questi parametri hanno come possibili valori i tipi del linguaggio.
  In particolare, possono assumere come valore qualsiasi tipo, esclusi i tipi primitivi (tipi base).

Tipi parametrici

- Un **tipo parametrico** è costituito dall'identificatore di una classe o di un'interfaccia, seguito, tra parentesi angolari, da un elenco di uno o più identificatori di classe, separati da virgole.
- Solitamente l'identificatore della classe rappresenta una raccolta, mentre il tipo degli elementi della raccolta è racchiuso tra parentesi angolari
- Un tipo parametrico viene chiamato anche "**tipo generico**"

Tipi parametrici

Vantaggi:

- Questo meccanismo consente di scrivere **codice più robusto** dal punto di vista dei tipi di dato (fornisce una migliore gestione del type checking durante la compilazione), evitando in molti casi il ricorso al casting da Object
- Esempio: realizzare una classe `Pair`, che rappresenta una coppia di oggetti dello stesso tipo.

Tipi parametrici

- In mancanza della programmazione parametrica (ad esempio, in Java 1.4) la classe si sarebbe dovuta realizzare secondo il seguente schema:

```
class Pair {
  private Object first, second;
  public Pair(Object a, Object b) { ...
  }
  public Object getFirst() { ... }
  public void setFirst(Object a) { ... }
  ...
}
```

Tipi parametrici

- Gli utenti della classe devono ricorrere al cast perché gli elementi estratti dalla coppia riacquistino il loro tipo originario, come nel seguente esempio:

```
Pair p = new Pair("uno", "due");
String a = (String) p.getFirst();
```

Tipi parametrici

- Vediamo ora come ovviare a questo problema rendendo la classe `Pair` parametrica:

```
class Pair<T> {
  private T first, second;
  public Pair(T a, T b) {
     first = a;
     second = b;
  }
  public T getFirst() { return first; }
  public void setFirst(T a) { first = a; }
  ...
}
```

Tipi parametrici

- La classe `Pair` ha un parametro di tipo, `T`
- I parametri di tipo vanno dichiarati dopo il nome della classe, racchiusi tra parentesi angolari
- Se vengono dichiarati più parametri di tipo, questi vanno separati da virgole
- All'interno della classe, un parametro di tipo si comporta (tranne poche eccezioni) come un tipo di dati vero e proprio
  - In particolare, un parametro di tipo si può usare come tipo di un campo, tipo di un parametro formale di un metodo e tipo di ritorno di un metodo

Tipi parametrici

- La nuova versione di `Pair` permette agli utenti della classe di specificare di che tipo di coppia si tratta e, così facendo, di evitare i cast:

```
Pair<String> p = new Pair<String>("uno","due");
String a = p.getFirst();
```

- Sia nella dichiarazione della variabile `p`, sia nell'instanziazione dell'oggetto `Pair` va indicato il parametro di tipo desiderato
- Come per i normali parametri dei metodi, `String` è il parametro attuale, che prende il posto del parametro formale `T` di `Pair`

Tipi parametrici

- Per compatibilità con le versioni precedenti di Java, è possibile usare una classe (o interfaccia) parametrica come se non lo fosse
- Quando utilizziamo una classe parametrica senza specificare i parametri di tipo, si dice che stiamo usando la **versione grezza** di quella classe
  - Es: l'interfaccia grezza Comparable della scorsa lezione!
- La versione grezza di queste classi permette alla nuova versione della libreria standard di essere compatibile con i programmi scritti con le versioni precedenti del linguaggio
  - Le classi grezze esistono solo per retro-compatibilità

Tipi parametrici

- Ad esempio, se `Pair` è la classe parametrica descritta nelle slide precedenti, è anche possibile utilizzarla così:

```
Pair p = new Pair("uno", "due");
String a = (String) p.getFirst();
```

- La prima riga provoca un warning in compilazione
- Il cast nella seconda riga è indispensabile

NB: Il codice nuovo dovrebbe sempre specificare i parametri di tipo delle classi parametriche

Tipi parametrici

- Esaminiamo un'ulteriore versione di `Pair`, in grado di contenere due oggetti di tipo diverso

```
class Pair<T, U> {  // due parametri tipo
  private T first;
  private U second;
  public Pair(T a, U b) {
      first = a; second = b;
  }
  public T getFirst() { return first; }
  public void setFirst(T a) { first = a; }
  public U getSecond() { return second; }
  public void setSecond(U a) { second = a; }
}
```

Tipi parametrici

- Si dice che una classe è parametrica se ha almeno un parametro di tipo
- Anche i singoli metodi e costruttori possono avere parametri di tipo, indipendentemente dal fatto che la classe cui appartengono sia parametrica o meno
  - I metodi statici non possono utilizzare i parametri di tipo della classe in cui sono contenuti
  - Il parametro di tipo va dichiarato prima del tipo restituito, racchiuso tra parentesi angolari
  - Questo parametro è visibile solo all'interno del metodo

Tipi parametrici

- Il seguente metodo parametrico restituisce l'elemento mediano (di posto intermedio) di un dato array

```
public static <T> T getMedian(T[] a) {
int l = a.length;
return a[l/2];
}
```

- In questo caso, il parametro di tipo permette di restituire un oggetto dello stesso tipo dell'array ricevuto come argomento

Tipi parametrici

- Quando si invoca un metodo parametrico, è opportuno, ma non obbligatorio, specificare il parametro di tipo attuale per quella chiamata
- Ad esempio, supponendo che il metodo `getMedian` appartenga ad una classe `Test`, lo si può invocare così:

```
String[] x = {"uno", "due", "tre"};
String s = Test.<String>getMedian(x);
```

- Il parametro attuale di tipo va quindi indicato tra il punto e il nome del metodo

Tipi parametrici

- È possibile omettere il parametro attuale di tipo. In questo caso, il compilatore cercherà di dedurre il tipo più appropriato, mediante un meccanismo chiamato *type inference* (inferenza di tipo)
- La type inference cerca di individuare il tipo più specifico che rende la chiamata corretta
- L'algoritmo di type inference non è né corretto né completo
- Le regole precise che il compilatore adotta nella type inference esulano dagli scopi di questo corso

Tipi parametrici

- Anche i costruttori possono essere parametrici, indipendentemente dal fatto che la loro classe sia parametrica o meno

```
Public class A<T> {
    Public <U> A(T x, U y) { ... }
    …
}
```

- In quest'esempio, il costruttore della classe parametrica `A` ha a sua volta un parametro di tipo chiamato `U`
- Mentre il parametro `T` è visibile in tutta la classe `A`, il parametro U è visibile solo all'interno di quel costruttore

Tipi parametrici

- Il costruttore in questione può essere invocato con la seguente sintassi

```
A<String> a =
new <Integer>A<String>("ciao", new Integer(100));
```

- Il parametro di tipo del costruttore (`Integer`) va specificato prima del nome della classe
- Il parametro di tipo della classe, come abbiamo già visto per la classe `Pair`, va specificato dopo il nome della classe

Tipi parametrici

- La programmazione parametrica dimostra tutta la sua utilità nella realizzazione di collezioni, ovvero classi deputate a contenere altri oggetti

```
public class ArrayList<E>
```

- Per creare un'istanza, `numberList`, della classe `ArrayList`, i cui elementi siano di tipo (referenza a) `Double`:

```
ArrayList<Double> numberList = new
        ArrayList<Double>();
```

Tipi parametrici

Esempi di uso:

- Inserimento

```
numberList.add(new Double(2.7));
numberList.add(2.7);
```

- Accesso:

```
Double wrapValue = numberList.get(0);
```

- Uso in un'espressione:

```
Sum = sum + wrapValue.doubleValue();
Sum = sum + wrapValue;
```

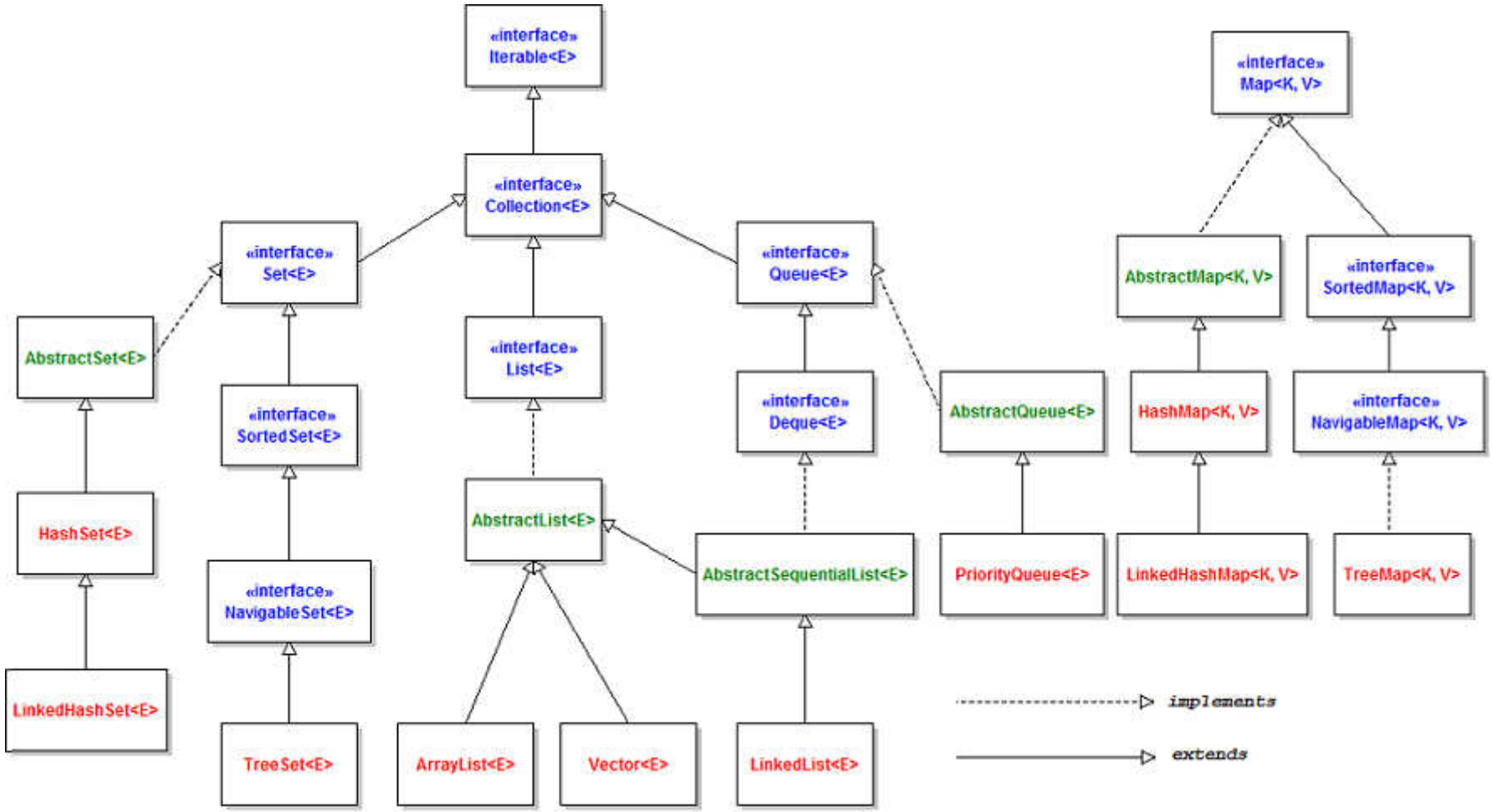
L'interfaccia `Collection`

- Il Java Collection Framework (JCF) è una parte della libreria standard dedicata alle collezioni
- Offre strutture dati di supporto molto utili alla programmazione, come array di dimensione dinamica, liste, insiemi, mappe associative (anche chiamate dizionari) e code
- Il JCF è costituito in pratica da una gerarchia che contiene classi astratte e interfacce ad ogni livello tranne l'ultimo, dove sono presenti soltanto classi che implementano interfacce e/o estendono classi astratte

L'interfaccia `Collection`

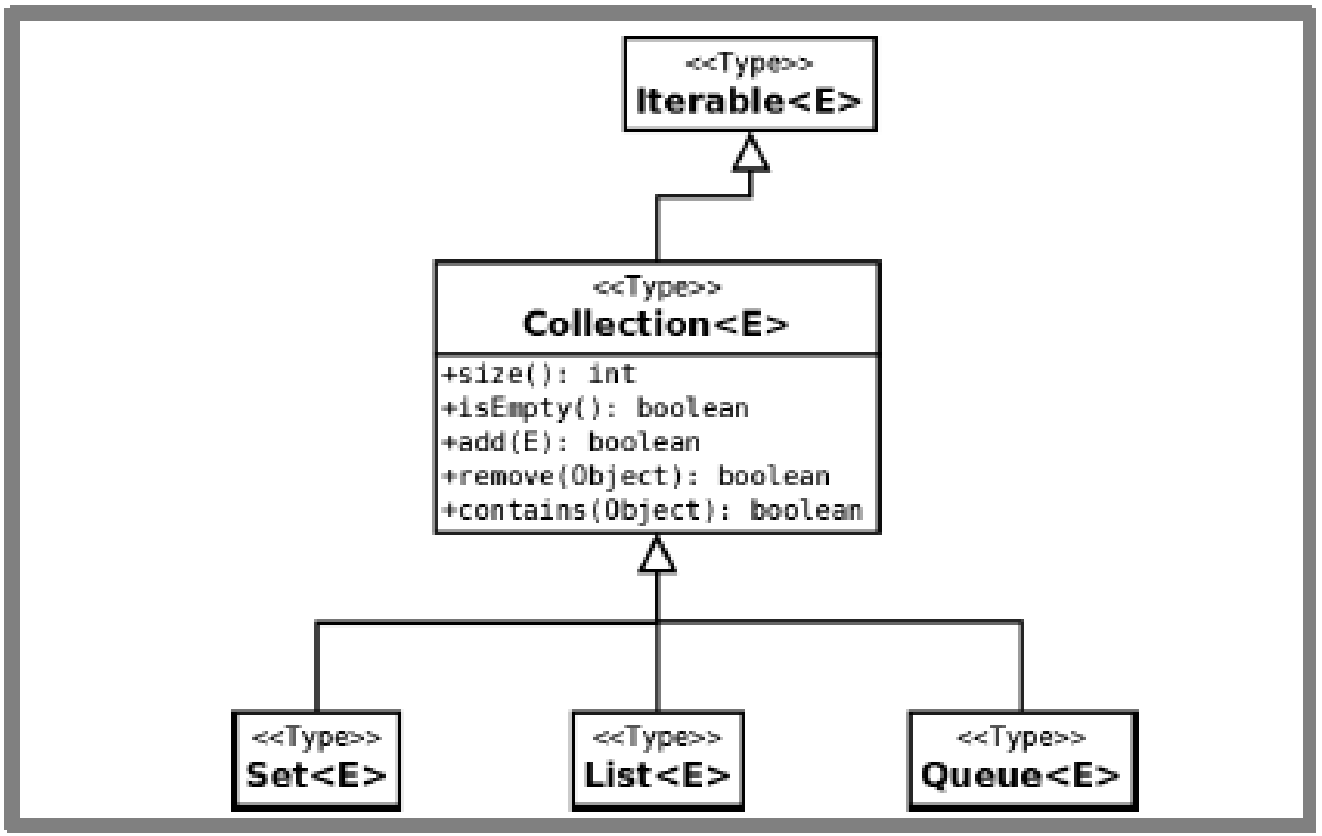
- In cima alla gerarchia troviamo le interfacce **`Collection`** e **`Map`**.
- L'interfaccia Collection estende la versione parametrica di **`Iterable`**.
- L'interfaccia Collection non va confusa con la classe Collection**s** che contiene numerosi algoritmi di supporto (metodi statici che operano su collezioni)
  - ad esempio, metodi che effettuano l'ordinamento

Overview: JCF



L'interfaccia `Collection`

- Interfacce collegate con `Collection`:



L'interfaccia `Collection`

- Può sorprendere che i metodi `contains` e `remove` accettino `Object` invece del tipo parametrico `E`
- Lo fanno perché non si corre alcun rischio a passare a questi due metodi un oggetto di tipo sbagliato
- Entrambi i metodi restituiranno `false`, senza nessun effetto sulla collezione stessa

Iteratori

- Consideriamo i seguenti esempi:
  - Dato un oggetto di una classe che implementa l'interfaccia `Collection` ("oggetto Collection" in breve) di studenti, visualizzare gli studenti migliori
  - Dato un oggetto `Collection` di membri di un club, aggiornare le quote dovute da ciascuno di essi
  - Dato un oggetto `Collection` di dipendenti a tempo pieno, calcolare il loro salario medio
- Notiamo che in ciascun esempio il compito da svolgere richiede l'accesso a tutti gli elementi di un oggetto `Collection`, uno dopo l'altro.

Iteratori

- Come si può fare in modo che qualsiasi implementazione dell'interfaccia `Collection` consenta ai suoi utilizzatori di eseguire un'iterazione che coinvolga, uno dopo l'altro, tutti gli elementi presenti in un suo esemplare, senza violare il principio di astrazione per i dati?
- La soluzione risiede nell'uso degli **iteratori**, oggetti che consentono di accedere agli elementi di oggetti `Collection`.

Iteratori: l'interfaccia `Iterator<E>`

- L'interfaccia `Iterator` astrae il processo di scandire gli elementi di una collezione uno alla volta
- Permette di scandire gli elementi della struttura dati a prescindere dall'implementazione della struttura dati
- In Java un iteratore ha due primitive fondamentali specificate dall'interfaccia **`java.util.Iterator`**:

1. `hasNext()`: verifica se c'è ancora un elemento nella collezione
2. `next()`: restituisce il prossimo elemento della collezione

Iteratori: l'interfaccia `Iterator<E>`

```
public interface Iterator<E> {
 public E next();
 public boolean hasNext();
 public void remove();
}
```

- Esempio: la classe `Scanner` implementa l'interfaccia `Iterator<String>`

Iteratori: l'interfaccia `Iterable<E>`

- L'interfaccia `Collection<E>` estende `Iterable<E>`

```
public interface Iterable<E> {
  public Iterator<E> iterator();
}
```

- Ci si aspetta dunque che ogni classe che implementa `Iterable<E>` abbia un metodo `iterator()` che restituisce un iteratore sugli elementi interni alla classe stessa

Iteratori: un esempio

- Esempio: sia `myColl` un riferimento ad un esemplare di una classe `Collection<String>`. Si vogliono visualizzare tutti i suoi elementi che iniziano con la lettera 'a'.
- Creiamo un oggetto iteratore. Un iteratore si ottiene invocando il metodo `iterator()` sull'oggetto che rappresenta la collezione stessa:

```
Iterator<String> itr = myColl.iterator();
```

Iteratori: un esempio

```
Iterator<String> itr = myColl.iterator();
```

- Eseguiamo la scansione:

```
String word;
while (itr.hasNext()){
  word=itr.next();
  if (word.charAt(0)==‘a’)
     System.out.println(word); }
```

Iteratori: schema tipico

```
Iterator<T> it = ottieni un iteratore
                 per la collezione
while (it.hasNext()){
  T elem = it.next();
  elabora l'elemento
}
```

Iteratori

- Ogni classe per rappresentare collezioni di elementi dovrebbe implementare l'interfaccia `Iterable`.
- Ad esempio l'interfaccia `java.util.List` estende l'interfaccia `Iterable`, pertanto ogni oggetto di tipo `List` è "iterabile".

Iteratori: il problema dei duplicati

```
public static boolean verificaDupOrdIterator(List S){
  Collections.sort(S);
  Iterator it=S.iterator();
  if (!it.hasNext()) return false;
  Object pred = it.next();
  while (it.hasNext()){
     Object succ=it.next();
     if (pred.equals(succ)) return true;
     pred=succ;
  }
  return false;
}
```

Il ciclo for-each

- Se un oggetto `x` appartiene ad una classe che implementa `Iterable<A>`, per una data classe `A`, è possibile scrivere il seguente ciclo for-each:

```
for (A a: x) {
// corpo del ciclo
...
}
```

Il ciclo for-each

- Il ciclo precedente è equivalente al blocco seguente:

```
Iterator<A> it = x.iterator();
while (it.hasNext()) {
A a = it.next();
// corpo del ciclo
...
}
```

- Come si vede, il ciclo for-each è più sintetico e riduce drasticamente il rischio di scrivere codice errato

Il ciclo `for-each`

- Il ciclo for-each:

```
for (A a: <exp>) {
    // corpo del ciclo
    ...
}
```

- è corretto a queste condizioni:

1. <exp> è una espressione di tipo “array di T” oppure di un sottotipo di “Iterable<T>”
2. T è assegnabile ad A

Il ciclo for-each

- Esempio: sia `myColl` un riferimento ad un esemplare di una classe `Collection<String>`. Si vogliono visualizzare tutti i suoi elementi che iniziano con la lettera 'a'.

- "*for each word in myColl…*"

```
for (String word: myColl)
   if (word.charAt(0)=='a')
     System.out.println(word);
```