

11

Generici e collezioni, pt 2

Mirko Viroli

mirko.viroli@unibo.it

C.D.L. Ingegneria e Scienze Informatiche
ALMA MATER STUDIORUM—Università di Bologna, Cesena

a.a. 2025/2026

Outline

Goal della lezione

- Approfondire alcuni concetti sui generici
- Presentare altre classi per le collezioni

Argomenti

- Il problema della type-erasure
- Polimorfismo vincolato
- Approfondimento sulle Wildcards
- Implementazioni di List e Map

Outline

- 1 Il problema della type-erasure
- 2 Polimorfismo vincolato
- 3 Java Wildcards e sostituibilità
- 4 Implementazioni di List
- 5 Altre classi: Arrays e Collections
- 6 Il caso delle java.util.Map
- 7 Un esercizio sulle collezioni

La classe generica List

```
1  /* Classe generica in X:  
2   - X è il tipo degli elementi della lista */  
3  public class List<X>{  
4  
5      private final X head;      // Testa della lista, tipo X  
6      private final List<X> tail;    // Coda della lista, tipo List<X>  
7  
8      public List(final X head, final List<X> tail){  
9          this.head = head;  
10         this.tail = tail;  
11     }  
12  
13     public X getHead(){  
14         return this.head;  
15     }  
16  
17     public List<X> getTail(){  
18         return this.tail;  
19     }  
20  
21     // getLength() e toString() invariate  
22     ...  
23 }
```

Uso di una classe generica

```
1 public class UseList{
2     public static void main(String[] s){
3         List<Integer> list = new List<Integer>(10, // Autoboxing
4             new List<Integer>(20,
5                 new List<Integer>(30,
6                     new List<Integer>(40,null))));  
7         // Cast NON necessari  
8         int first = list.getHead(); // Unboxing  
9         int second = list.getTail().getHead();  
10        int third = list.getTail().getTail().getHead();  
11        System.out.println(first+ " "+second+ " "+third);  
12        System.out.println(list.toString());  
13        System.out.println(list.getLength());  
14  
15        // Usabile anche con le stringhe  
16        List<String> list2 = new List<String>("a",
17            new List<String>("b",
18                new List<String>("c",
19                    new List<String>("d",null))));  
20        System.out.println(list2.toString());  
21    }  
22}
```

Terminologia, e elementi essenziali

Data una classe generica $C<X, Y>$..

- X e Y sono dette le sue **type-variable**
- X e Y possono essere usate come un qualunque tipo dentro la classe (con alcune limitazioni che vedremo)

I clienti delle classi generiche

- Devono usare **tipi generici**: versioni “istanziate” delle classi generiche
 - ▶ $C<String, Integer>$, $C<C<Object, Object>, Object>$
 - ▶ Non C senza parametri, altrimenti vengono segnalati dei warning
- Ogni type-variable va sostituita con un tipo effettivo, ossia con un **parametro**, che può essere
 - ▶ una classe (non-generica), p.e. `Object`, `String`...
 - ▶ una type-variable definita, p.e. X, Y (usate dentro la classe $C<X, Y>$)
 - ▶ un tipo generico completamente istanziato, p.e. $C<Object, Object>$
 - ▶ ..o parzialmente istanziato, p.e. $C<Object, X>$ (in $C<X, Y>$)
 - ▶ NON con un tipo primitivo

Limitazioni all'uso dei generici

Una type-variable (X) non è usabile in istruzioni del tipo:

- `new X()`, `new X[]{}..`, `new X[10]`, `instanceof X`
- Il compilatore segnala un errore
- errore anche l'`instanceof` su un tipo generico:
 - `instanceof C<String>`

Type-variable, e tipi generici danno warning se usati in situazioni “borderline”

- ... (X)o, (C<String>)o
- il compilatore segnala un “unchecked warning”

Perché queste limitazioni? (Odersky & Runne & Wadler, 1998)

- Derivano dallo schema di supporto ad erasure
- Filosofia: quando la semantica non è quella attesa dall'utente si preferisce dare errore di compilazione, usando il warning solo per operazioni necessarie

Qualche prova

```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 class P<X> {
5     void m() {
6         X x = (X)new Object(); // warning
7         // X x = new X(); // error
8         X[] a = (X[])new Object[10]; // warning
9         // X[] a = new X[10]; // error
10        // boolean b = new Object() instanceof X; // error
11    }
12 }
13
14 public class ErasurePitfalls {
15
16     public static void main(String[] args) {
17         Object o = (List<String>)new Object();
18         // boolean b = new Object() instanceof List<String> // error
19         // Object o = new List<Pair>[10]; // error
20     }
21
22 }
```

La classe generica Vector

Un dettaglio della sua implementazione:

Per via della type erasure, il suo campo non può essere di tipo X[], bensì Object[]

```
1 public class Vector<X>{
2 // X è la type-variable, ossia il tipo degli elementi
3 ...
4     public Vector(){...}
5
6     public void addElement(X e){...} // Input di tipo X
7
8     public X getElementAt(int pos){...} // Output di tipo X
9
10    public int getLength(){...}
11
12    public String toString(){...}
13 }
```

Uso di Vector<X>

```
1 public class UseVector{  
2     public static void main(String[] s){  
3  
4         // Il tipo di vs è Vector<String>  
5         // Ma la sua classe è Vector<X>  
6         final Vector<String> vs = new Vector<String>();  
7         vs.addElement("Prova");  
8         vs.addElement("di");  
9         vs.addElement("Vettore");  
10        final String str = vs.elementAt(0) + " " +  
11            vs.elementAt(1) + " " +  
12            vs.elementAt(2); // Nota, nessun cast!  
13        System.out.println(str);  
14  
15        final Vector<Integer> vi=new Vector<Integer>();  
16        vi.addElement(10); // Autoboxing  
17        vi.addElement(20);  
18        vi.addElement(30);  
19        final int i = vi.elementAt(0) + // Unboxing  
20            vi.elementAt(1) +  
21            vi.elementAt(2);  
22        System.out.println(i);  
23    }  
}
```

Implementazione di Vector pt 1

```
1 public class Vector<X>{
2
3     private final static int INITIAL_SIZE = 10;
4
5     private Object[] elements; // Deposito elementi, non posso usare X[]!!!
6     private int size; // Numero di elementi
7
8     public Vector(){ // Inizialmente vuoto
9         this.elements = new Object[INITIAL_SIZE];
10        this.size = 0;
11    }
12
13    public void addElement(final X e){
14        if (this.size == elements.length){
15            this.expand(); // Se non c'è spazio
16        }
17        this.elements[this.size] = e;
18        this.size++;
19    }
20
21    public X getElementAt(final int position){
22        // unchecked warning
23        return (X)this.elements[position];
24    }
}
```

Implementazione di Vector pt 2

```
1  public int getLength(){
2      return this.size;
3  }
4
5
6  private void expand(){ // Raddoppio lo spazio..
7      final Object[] newElements = new Object[this.elements.length*2];
8      for (int i=0; i < this.elements.length; i++){
9          newElements[i] = this.elements[i];
10     }
11     this.elements = newElements;
12 }
13
14 public String toString(){
15     String s="|";
16     for (int i=0; i < size; i++){
17         s = s + this.elements[i] + "|";
18     }
19     return s;
20 }
21 }
```

Ancora sugli “unchecked warning” coi generici

```
1 public class ShowCast {  
2  
3     public static void main(String[] args) {  
4         Vector<String> v = new Vector<>();  
5         v.addElement("a");  
6         v.addElement("b");  
7         v.addElement("c");  
8         Object o = v;  
9         Vector<Integer> v2 = (Vector<Integer>)o; // unchecked warning  
10        // Da qui in poi siamo in situazione "pericolosa"  
11        // Cosa può succedere?  
12        System.out.println(v2.elementAt(0));  
13        System.out.println(v2.elementAt(0).intValue());  
14    }  
15 }
```

Unchecked warning e “unsafety”

- un unchecked warning per via di conversione con genericità espone l'esecuzione dei nostri programmi a possibili errori successivi (così come i downcast)
- usarli solo se si è certi ciò non succeda (in tal caso vedremo come “silenziare il warning”)

Outline

- 1 Il problema della type-erasure
- 2 Polimorfismo vincolato
- 3 Java Wildcards e sostituibilità
- 4 Implementazioni di List
- 5 Altre classi: Arrays e Collections
- 6 Il caso delle java.util.Map
- 7 Un esercizio sulle collezioni

Polimorfismo vincolato

Negli esempi visti finora..

- Data una classe `C<X>`, X può essere istanziato a qualunque sottotipo di `Object`
- In effetti la definizione `class C<X>{...}` equivale a
`class C<X extends Object>{...}`

Polimorfismo vincolato (Igarashi & Pierce & Wadler, 2000)

- In generale invece, può essere opportuno vincolare in qualche modo le possibili istanziazioni di X, ad essere sottotipo di un tipo più specifico di `Object`
- `class C<X extends D>{...}`
- In tal caso, dentro C, si potrà assumere che gli oggetti di tipo X rispondano ai metodi della classe D

LampsRow generica: Definizione

```
1 public class LampsRow<L extends SimpleLamp> {
2
3     private Vector<L> lamps;
4
5     public LampsRow(){
6         this.lamps = new Vector<>(); // inferenza
7     }
8
9     public L getLamp(int position){
10        return this.lamps.getElementAt(position);
11    }
12
13    public void addLamp(L lamp){
14        this.lamps.addElement(lamp);
15    }
16
17    public void switchOffAll(){
18        for (int i = 0; i < lamps.getLength(); i++){
19            this.getLamp(i).switchOff();
20        }
21    }
22
23    public String toString(){
24        return this.lamps.toString();
25    }
26}
```

Uso

Motivazione per questa genericità. Ha senso se:

- si ritiene molto frequente l'uso di SimpleLamp simili tra loro, ossia di una comune specializzazione (classe)
- è frequente l'uso di getLamp() e quindi del cast

```
1 public class UseLampsRow {  
2  
3     public static void main(String[] s){  
4         final LampsRow<UnlimitedLamp> lr = new LampsRow<>();  
5         lr.addLamp(new UnlimitedLamp());  
6         lr.addLamp(new UnlimitedLamp());  
7         lr.addLamp(new UnlimitedLamp());  
8  
9         lr.getLamp(0).switchOn();  
10        lr.switchOffAll();  
11  
12        System.out.println(lr.getLamp(0).isOver());  
13        System.out.println(lr);  
14    }  
15}
```

Outline

1 Il problema della type-erasure

2 Polimorfismo vincolato

3 Java Wildcards e sostituibilità

4 Implementazioni di List

5 Altre classi: Arrays e Collections

6 Il caso delle java.util.Map

7 Un esercizio sulle collezioni

Java Wildcards

```
1 // Gerarchia dei wrapper Numbers in java.lang
2 abstract class Number extends Object{...}
3 class Integer extends Number{...}
4 class Double extends Number{...}
5 class Long extends Number{...}
6 class Float extends Number{...}
7 ...
```

```
1 // Accetta qualunque Vector<T> con T <: Number
2 // Vector<Integer>, Vector<Double>, Vector<Float>,...
3 void m(Vector<? extends Number> arg){...}
4
5 // Accetta qualunque Vector<T>
6 void m(Vector<?> arg){...}
7
8 // Accetta qualunque Vector<T> con Integer <: T
9 // Vector<Integer>, Vector<Number>, e Vector<Object> solo!
10 void m(Vector<? super Integer> arg){...}
```

Java Wildcards

3 tipi di wildcard (Igarashi & Viroli, 2002)

- Bounded (covariante): `C<? extends T>`
 - ▶ accetta un qualunque `C<S>` con `S <: T`
- Bounded (controvariante): `C<? super T>`
 - ▶ accetta un qualunque `C<S>` con `S >: T`
- Unbounded: `C<?>`
 - ▶ accetta un qualunque `C<S>`

Uso delle librerie che dichiarano tipi wildcard

- Piuttosto semplice, basta passare un argomento compatibile o si ha un errore a tempo di compilazione

Progettazione di librerie che usano tipi wildcard

- Molto avanzato: le wildcard pongono limiti alle operazioni che uno può eseguire, derivanti dal principio di sostituibilità

Approfondimento: sulla sostituibilità dei generici

Domanda: `Vector<Integer>` è un sottotipo di `Vector<Object>?`

Ovvero, possiamo pensare di passare un `Vector<Integer>` in tutti i contesti in cui invece ci si aspetta un `Vector<Object>?`

Risposta: no!! Sembra di sì.. ma:

cosa succede se nel metodo qui sotto passiamo un `Vector<Integer>?`

⇒ potremmo facilmente compromettere l'integrità del vettore

```
1 void addAString(Vector<Object> vector){  
2     vector.addElement("warning!");  
3 }  
4 ...  
5 Vector<Integer> vec=new Vector<>(); // Inferenza  
6 vec.addElement(new Integer(0));  
7 vec.addElement(new Integer(1));  
8 vec.addElement(new Integer(2));  
9 addAString(vec); // ATTENZIONE!!  
10 int n = vec.getElementAt(3).intValue(); // break!
```

Subtyping e safety

Safety di un linguaggio OO

Se nessuna combinazione di istruzioni porta a poter invocare un metodo su un oggetto la cui classe non lo definisce

- È necessario che il subtyping segua il principio di sostituibilità
- In Java bisogna evitare gli usi di downcast e unchecked cast

Più in generale, se non possono accadere errori a tempo di esecuzione...

Java

- Si pone dove possibile l'obiettivo della safety
- Quindi, non è vero che `Vector<Integer> <: Vector<Object>`

Generici e safety

In generale, istanziazioni diverse di una classe generica sono scollegate

- non c'è **covarianza**: non è vero che `C<T> <: C<S>` con `T <: S`
- non c'è **controvarianza**: non è vero che `C<S> <: C<T>` con `T <: S`

Unsafety con gli array di Java

Fin da Java 1, gli array di Java sono stati trattati come covarianti!

- Gli array assomigliano moltissimo ad un tipo generico
- Integer[] ~ Array<Integer>, T[] ~ Array<T>
- E quindi sappiamo che non sarebbe safe gestirli con covarianza
- E invece in Java è esattamente così!! P.e. Integer[] <: Object[]
- Quindi ogni scrittura su array potrebbe successivamente potenzialmente fallire... lanciando un ArrayStoreException

```
1 Object[] o = new Integer[]{1,2,3}; // OK per covarianza
2 o[0] = "a"; // Lancia ArrayStoreException
3 // per prevenire la unsafety di successive istruzioni come:
4 // Integer[] ar=((Integer[])o);
5 // int i=ar[0].intValue();
```

Covarianza e operazioni di accesso

La covarianza ($C<T> <: C<S>$ con $T <: S$) sarebbe ammisible se:

- La classe $C<X>$ non avesse operazioni che ricevono oggetti X
- Ossia, ha solo campi privati e nessun metodo con argomento X

La controvarianza ($C<S> <: C<T>$ con $T <: S$) sarebbe ammisible se:

- La classe $C<X>$ non avesse operazioni che producono oggetti X
- Ossia, ha solo campi privati e nessun metodo con tipo di ritorno X

In pratica:

- La maggior parte delle classi generiche $C<X>$ hanno campi di tipo X (composizione) e operazioni getter e setter, e quindi per loro covarianza e controvarianza non funzionano

I wildcard types:

- Sono delle “interfacce” soprattutto dei tipi generici, per le quali invece vale co-/contro-varianza

Esempio Wildcard Bounded covariante: C<? extends T>

```
1 public class Cov{  
2  
3     public static void printAll(Vector<? extends Number> vect){  
4         // Questo metodo usa solo 'X getElementAt()'  
5         // Quindi l'input può essere covariante!  
6         // Si noti che Number definisce 'int intValue()'  
7         for (int i=0;i<vect.getLength();i++){  
8             System.out.println(vect.getElementAt(i).intValue());  
9         }  
10    }  
11  
12    public static void main(String[] s){  
13        Vector<Integer> vector = new Vector<>();  
14        vector.addElement(1);  
15        vector.addElement(2);  
16        vector.addElement(3);  
17        // Posso passare Vector<Integer> dove si attende  
18        // Vector<? extends Number>  
19        printAll(vector);  
20    }  
21 }
```

Esempio Wildcard Bounded controvariante:

C<? super T>

```
1 public class Contra{  
2  
3     public static void addStrings(Vector<? super String> vect,  
4         String s, int n){  
5         // Questo metodo usa solo 'void addElement(X x)',  
6         // Quindi l'input può essere contra-variante!  
7         for (int i=0;i<n;i++){  
8             vect.addElement(s);  
9         }  
10    }  
11  
12    public static void main(String[] s){  
13        Vector<Object> vector = new Vector<>();  
14        vector.addElement(1);  
15        vector.addElement(new java.util.Date());  
16        // Posso passare Vector<Object> dove si attende  
17        // Vector<? super String>  
18        addStrings(vector, "aggiunta", 10);  
19        System.out.println(vector);  
20    }  
}
```

Esempio Wildcard Unbounded: C<?>

```
1 public class Unbounded{  
2  
3     public static void printLength(Vector<?> vect){  
4         // Questo metodo usa solo 'int getLength()'  
5         // Quindi l'input può essere bi-variante!  
6         System.out.println(vect.getLength());  
7     }  
8  
9     public static void main(String[] s){  
10        Vector<Integer> vector = new Vector<>();  
11        vector.addElement(1);  
12        vector.addElement(2);  
13        vector.addElement(3);  
14        // Posso passare Vector<Integer> dove si attende  
15        // Vector<?>  
16        printLength(vector);  
17    }  
18}
```

Un uso tipico: metodi in classi generiche

```
1 public class Vector<X>{  
2     ...  
3     public void addAllFrom(Vector<? extends X> vect){  
4         for (int i=0;i<vect.getLength();i++){  
5             this.addElement(vect.getElementAt(i));  
6         }  
7     }  
8 }
```

```
1 public class UseVector{  
2     public static void main(String[] s){  
3         Vector<Object> vo = new Vector<>();  
4         vo.addElement(1);  
5         vo.addElement("2");  
6         vo.addElement(new java.util.Date());  
7  
8         Vector<Double> vd = new Vector<>();  
9         vd.addElement(Math.random());  
10        vd.addElement(Math.random());  
11  
12        vo.addAll(vd);  
13        System.out.println(vo);  
14    }  
15 }
```

Un uso tipico: metodi in classi generiche

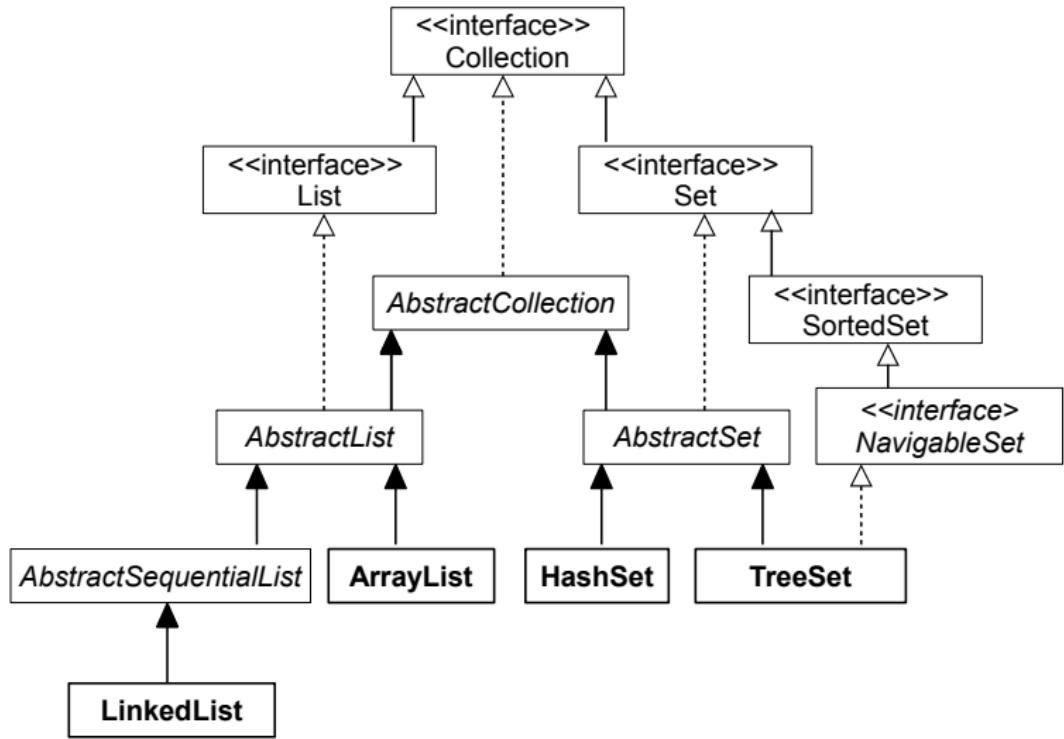
```
1 public class Vector<X>{  
2     ...  
3     public void addAllFrom(Vector<? extends X> vect){  
4         for (int i=0;i<vect.getLength();i++){  
5             this.addElement(vect.getElementAt(i));  
6         }  
7     }  
8 }
```

```
1 public class UseVector{  
2     public static void main(String[] s){  
3         Vector<Object> vo = new Vector<>();  
4         vo.addElement(1);  
5         vo.addElement("2");  
6         vo.addElement(new java.util.Date());  
7  
8         Vector<Double> vd = new Vector<>();  
9         vd.addElement(Math.random());  
10        vd.addElement(Math.random());  
11  
12        vo.addAll(vd);  
13        System.out.println(vo);  
14    }  
15 }
```

Outline

- 1 Il problema della type-erasure
- 2 Polimorfismo vincolato
- 3 Java Wildcards e sostituibilità
- 4 Implementazioni di List
- 5 Altre classi: Arrays e Collections
- 6 Il caso delle java.util.Map
- 7 Un esercizio sulle collezioni

Implementazione collezioni – UML



List

```
1 public interface List<E> extends Collection<E> {
2     // Additional Bulk Operations
3     // aggiunge gli elementi in pos. index
4     boolean addAll(int index, Collection<? extends E> c);
5
6     // Positional Access Operations
7     E get(int index);
8     E set(int index, E element);
9     void add(int index, E element);
10    E remove(int index);
11
12    // Search Operations
13    int indexOf(Object o);    // basato su Object.equals
14    int lastIndexOf(Object o); // basato su Object.equals
15
16    // List Iterators
17    ListIterator<E> listIterator();
18    ListIterator<E> listIterator(int index);
19
20    // View
21    List<E> subList(int fromIndex, int toIndex);
22}
```

Implementazioni di List

Caratteristiche delle liste

- Sono sequenze: ogni elemento ha una posizione
- Il problema fondamentale è realizzare i metodi posizionali in modo efficiente, considerando il fatto che la lista può modificarsi nel tempo (altrimenti andrebbe bene un array)

Approccio 1: ArrayList

Internamente usa un array di elementi con capacità maggiore di quella al momento necessaria. Se serve ulteriore spazio si alloca trasparentemente un nuovo e più grande array

Approccio 2: LinkedList

Usa una double-linked list. L'oggetto `LinkedList` mantiene il riferimento al primo e ultimo elemento della lista, e alla dimensione della lista

ArrayList

Caratteristiche di performance

- Lettura/scrittura in data posizione sono a tempo costante
- La `add()` è tempo costante ammortizzato, ossia, n add si effettuano in $O(n)$
- Tutte le altre operazioni sono a tempo lineare

Funzionalità aggiuntive

Per migliorare le performance (e l'occupazione in memoria) in taluni casi l'utente esperto può usare funzioni aggiuntive

- Specificare la dim iniziale dell'array interno nella `new`
- `trimToSize()` e `ensureCapacity()` per modifiche in itinere

ArrayList: aspetti aggiuntivi

```
1 public class ArrayList<E> extends AbstractList<E>
2     implements List<E>, RandomAccess, Cloneable, java.io.
3 Serializable
4 {
5     public ArrayList(int initialCapacity) {...}
6
7     // Usa un valore di default per la capacità iniziale (10)
8     public ArrayList() {...}
9
10    // Riempe coi valori di c
11    public ArrayList(Collection<? extends E> c) {...}
12
13    // Riduce la dimensione dell'array interno
14    public void trimToSize() {...}
15
16    // Aumenta la dimensione dell'array interno
17    public void ensureCapacity(int minCapacity) {...}
18}
```

UseArrayList

```
1 public class UseArrayList{
2
3     public static void main(String[] s){
4
5         final ArrayList<Persona> alist = new ArrayList<>();
6         alist.ensureCapacity(30); // per performance
7         for (int anno=1960; anno<1970; anno++){
8             alist.add(new Persona("Rossi",anno,false));
9             alist.add(new Persona("Bianchi",anno,true));
10            alist.add(new Persona("Verdi",anno,false));
11        }
12        final Persona p = new Persona("Rossi",1967,false);
13        int pos = alist.indexOf(p);
14        System.out.println(p+" in position "+pos);
15
16        // Iteratore da pos fino in fondo.. lo uso per eliminare
17        final ListIterator<Persona> iterator = alist.listIterator(pos);
18        while (iterator.hasNext()){
19            iterator.next();
20            iterator.remove();
21        }
22        for (final Persona p2: alist){
23            System.out.println(alist.indexOf(p2)+"\t"+p2);
24        }
25        alist.trimToSize(); // riduco le dimensioni
26    }
27}
```

LinkedList

Caratteristiche di performance

- Accesso e modifica in una data posizione hanno costo lineare
- Operazioni in testa o coda, quindi, sono a tempo costante
- Usa in generale meglio la memoria rispetto ArrayList
- (di norma però si preferisce ArrayList)

Caratteristiche aggiuntive

- Implementa anche l'interfaccia Queue, che modella una coda (ad esempio FIFO)
- Implementa anche l'interfaccia Deque che estende Queue, una coda a due estremi

LinkedList: funzioni aggiuntive relative a code (e stack)

```
1 public interface Queue<E> extends Collection<E> {
2     boolean offer(E e); // inserisce se c'è spazio
3     E poll(); // preleva se c'è il primo
4     E element(); // legge se c'è il primo, altrimenti eccezione
5     E peek(); // legge se c'è il primo, altrimenti null
6 }
```

```
1 public interface Deque<E> extends Queue<E> {
2     void addFirst(E e);
3     void addLast(E e);
4     boolean offerFirst(E e);
5     boolean offerLast(E e);
6     E removeFirst();
7     E removeLast();
8     E pollFirst();
9     E pollLast();
10    E getFirst();
11    E getLast();
12    E peekFirst();
13    E peekLast();
14    boolean removeFirstOccurrence(Object o);
15    boolean removeLastOccurrence(Object o);
16    void push(E e); // identical to addFirst()
17    E pop(); // identical to removeFirst()
18 }
```

LinkedList: costruzione

```
1 public class LinkedList<E>
2     extends AbstractSequentialList<E>
3     implements List<E>, Deque<E>, Cloneable, java.io.Serializable {
4
5     public LinkedList() {...}
6
7     public LinkedList(Collection<? extends E> c) {...}
8 }
```

UseLinkedList

```
1 public class UseLinkedList{  
2  
3     private static final String ELEMS = "A B C D E F G H I L M";  
4  
5     public static void main(String[] s){  
6         final LinkedList<String> llist =  
7             new LinkedList<>(Arrays.asList(ELEMS.split(" ")));  
8         llist.addFirst("*");  
9         llist.addLast("*");  
10        llist.removeFirstOccurrence("*");  
11        llist.removeLastOccurrence("*");  
12  
13        // bad performance  
14        llist.add(llist.indexOf("L"), "K");  
15        // better performance  
16        final ListIterator<String> it = llist.listIterator();  
17        while (it.hasNext()) {  
18            if (it.next().equals("I")) {  
19                it.add("J");  
20                break;  
21            }  
22        }  
23        final String[] str = llist.toArray(new String[0]);  
24        System.out.println(Arrays.toString(str));  
25    }  
}
```

Outline

- 1 Il problema della type-erasure
- 2 Polimorfismo vincolato
- 3 Java Wildcards e sostituibilità
- 4 Implementazioni di List
- 5 Altre classi: **Arrays e Collections**
- 6 Il caso delle java.util.Map
- 7 Un esercizio sulle collezioni

Classi di utilità (moduli): Arrays e Collections

java.util.Arrays

- Contiene varie funzionalità d'ausilio alla gestione degli array
- In genere ha varie versione dei metodi per ogni array di tipo primitivo
- Ricerca binaria (dicotomica), Ordinamento (quicksort), copia
- Operazioni base (`toString`, `equals`, `hashCode`), anche ricorsive

java.util.Collections

- Raccoglie metodi statici che sarebbero potuti appartenere alle varie classi/interfacce viste
- Ricerca binaria (dicotomica), Ordinamento (quicksort), copia, min, max, sublist, replace, reverse, rotate, shuffle
- Wrapper immutabili: `unmodifiableList`, `unmodifiableSet`
- Con esempi notevoli d'uso delle wildcard

Arrays: qualche esempio di metodi

```
1 public class Arrays {  
2     public static void sort(Object[] a, int fromIndex, int toIndex) {...}  
3     public static <T> void sort(T[] a, Comparator<? super T> c) {...}  
4     ...  
5     public static int binarySearch(int[] a, int key) {...}  
6     public static int binarySearch(char[] a, char key) {...}  
7     public static <T> int binarySearch(T[] a, T key, Comparator<? super T> c  
8 ) {...}  
9     ...  
10    public static <T> List<T> asList(T... a) {...}  
11    public static <T> T[] copyOfRange(T[] original, int from, int to) {...}  
12    public static void fill(Object[] a, int from, int to, Object val) {...}  
13  
14    public static boolean deepEquals(Object[] a1, Object[] a2) {...}  
15    public static int deepHashCode(Object a[]) {...}  
16    public static String deepToString(Object[] a) {...}  
17  
18    public static String toString(long[] a) {...}  
19    public static String toString(int[] a) {...}  
20    public static String toString(Object[] a) {...}  
21 }  
22 }
```

UseArrays: qualche esempio di applicazione

```
1 public class UseArrays implements Comparator<Integer>{
2
3     public int compare(Integer a, Integer b){
4         return b-a; // ordine inverso
5     }
6
7     public static void main(String[] s){
8
9         String[] array = new String[] {"a", "b", "c", "d", "e", "f"};
10
11        List<String> list = Arrays.asList("a", "b", "c", "d", "e", "f");
12
13        final Integer[] a = new Integer[20];
14        for (int i=0;i<20;i++){
15            a[i] = (int)(Math.random()*100);
16        }
17        System.out.println("rand: "+Arrays.toString(a));
18        Arrays.sort(a); // sort in ordine naturale
19        System.out.println("sort1: "+Arrays.toString(a));
20        Arrays.sort(a, new UseArrays()); // sort col comparator
21        System.out.println("sort2: "+Arrays.toString(a));
22        Arrays.fill(a, 10, 15, 0); // fill nel range
23        System.out.println("fill: "+Arrays.toString(a));
24
25        final Integer[][] b = new Integer[10][];
26        Arrays.fill(b, a); // fill di un array di array
27        System.out.println("recu: "+Arrays.deepToString(b));
28    }
}
```

Collections: qualche esempio di metodi

```
1 // nota: tutti metodi public static!!
2 public class Collections {
3     // ordinamenti e varie
4     <T extends Comparable<? super T>> void sort(List<T> list) {...}
5     <T> void sort(List<T> list, Comparator<? super T> c) {...}
6     <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)
7     <T> T min(Collection<? extends T> coll, Comparator<? super T> comp)
8
9     // modifiche
10    void reverse(List<?> list) {...}
11    void shuffle(List<?> list) {...}
12    <T> void fill(List<? super T> list, T obj) {...}
13    <T> void copy(List<? super T> dest, List<? extends T> src) {...}
14
15    // letture varie
16    int indexOfSubList(List<?> source, List<?> target) {...}
17    boolean disjoint(Collection<?> c1, Collection<?> c2) {...}
18    int frequency(Collection<?> c, Object o) {...}
19
20    // costruzioni di collezioni
21    <T> List<T> emptyList() {...}
22    <T> Set<T> emptySet() {...}
23    <T> List<T> nCopies(int n, T o) {...}
24    <T> Set<T> singleton(T o) {...}
25    <T> List<T> singletonList(T o) {...}
26}
```

UseCollections: qualche esempio di applicazione

```
1 public class UseCollections{  
2  
3     public static void main(String[] s){  
4         final List<Integer> list = Arrays.asList(new Integer[]{0,1,2,3,4});  
5         final Set<List<Integer>> set = new HashSet<>();  
6  
7         for (int i=0;i<5;i++){  
8             final List<Integer> l2 = new ArrayList<>(list);  
9             Collections.shuffle(l2); // permuto  
10            if (!set.contains(l2)){ // no duplicazione!  
11                set.add(l2); // aggiungo  
12            }  
13        }  
14        System.out.println("shuf: "+set); // [[4,1,2,3,0],[3,1,4,0,2],...  
15  
16        int ct=0;  
17        for (final List<Integer> l:set){  
18            Collections.fill(l,ct++);  
19        }  
20        System.out.println("inc: "+set); // [[0,0,0,0,0],[1,1,1,1,1],...  
21  
22        System.out.println("cop: "+Collections.nCopies(5,list));  
23        // [[0,1,2,3,4],[0,1,2,3,4],...  
24    }  
25 }
```

Outline

1 Il problema della type-erasure

2 Polimorfismo vincolato

3 Java Wildcards e sostituibilità

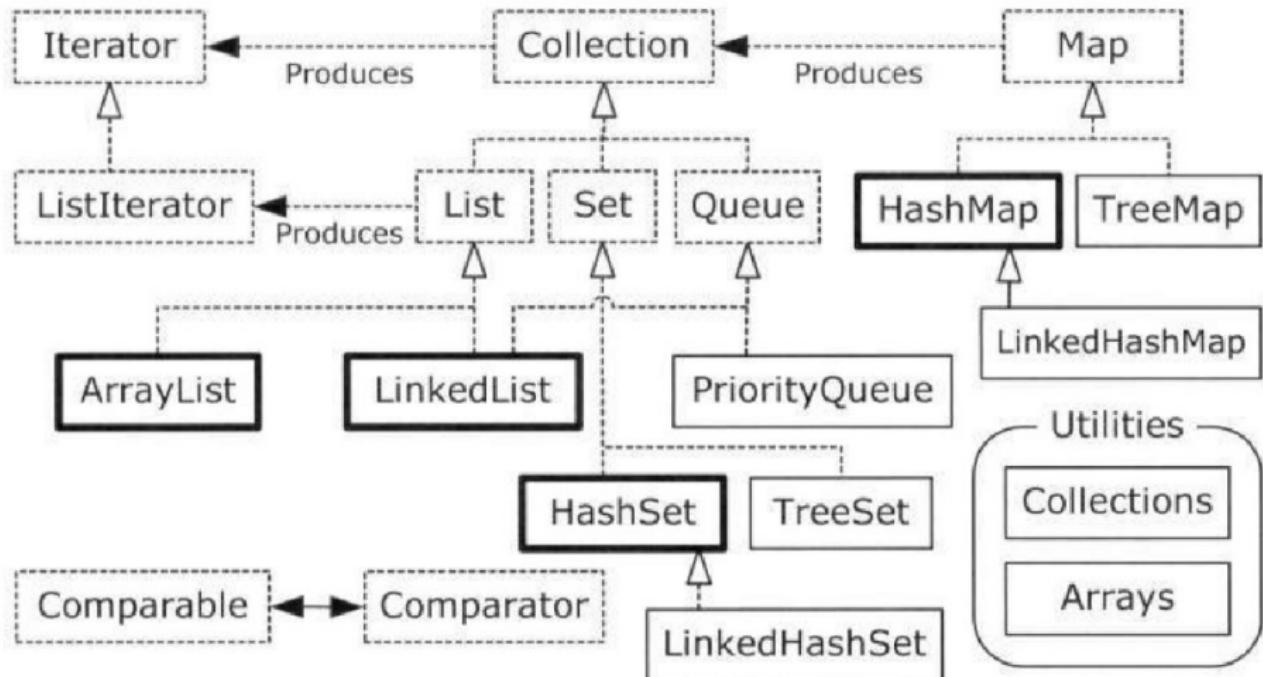
4 Implementazioni di List

5 Altre classi: Arrays e Collections

6 Il caso delle `java.util.Map`

7 Un esercizio sulle collezioni

JCF – struttura semplificata



Map

```
1 public interface Map<K,V> {  
2  
3     // Query Operations  
4     int size();  
5     boolean isEmpty();  
6     boolean containsKey(Object key);           // usa Object.equals  
7     boolean containsValue(Object value);        // usa Object.equals  
8     V get(Object key);                         // accesso a valore  
9  
10    // Modification Operations  
11    V put(K key, V value);                    // inserimento chiave-valore  
12    V remove(Object key);                     // rimozione chiave(-valore)  
13  
14    // Bulk Operations  
15    void putAll(Map<? extends K, ? extends V> m);  
16    void clear();                            // cancella tutti  
17  
18    // Views  
19    Set<K> keySet();                        // set di valori  
20    Collection<V> values();                 // collezione di chiavi  
21  
22    ... // qualche altro  
23 }
```

Usare le mappe

```
1 public class UseMap {  
2  
3     public static void main(String[] args) {  
4         // Uso una incarnazione, ma poi lavoro sull'interfaccia  
5         final Map<Integer, String> map = new HashMap<>();  
6         // Una mappa è una funzione discreta  
7         map.put(345211, "Bianchi");  
8         map.put(345122, "Rossi");  
9         map.put(243001, "Verdi");  
10        System.out.println(map); // {345211=Bianchi, 243001=Verdi, 345122=Rossi}  
11        map.put(243001, "Neri"); // Rimpiazza Verdi  
12  
13        // modo poco prestante per accedere alle coppie chiave-valore  
14        for (final Integer i : map.keySet()) {  
15            System.out.println("Chiave: " + i + " Valore: " + map.get(i));  
16        }  
17        // modo prestante per accedere ai soli valori  
18        for (final String s : map.values()) {  
19            System.out.println("Valore: " + s);  
20        }  
21    }  
22}
```

Due implementazioni di Map e AbstractMap

Map<K, V>

- Rappresenta una funzione dal dominio K in V
- La mappa tiene tutte le associazioni (o “entry”)
- Non posso esistere due entry con stessa chiave (`Object.equals`)

HashMap

- Sostanzialmente un HashSet di coppie Key, Value
- L'accesso ad un valore tramite la chiave è fatto con hashing
- Accesso a tempo costante, a discapito di overhead in memoria

TreeMap

- Sostanzialmente un TreeSet di coppie Key, Value
- L'accesso ad un valore tramite la chiave è fatto con red-black tree
- Accesso in tempo logaritmico
- Le chiavi devono essere ordinate, come per i TreeSet

Outline

1 Il problema della type-erasure

2 Polimorfismo vincolato

3 Java Wildcards e sostituibilità

4 Implementazioni di List

5 Altre classi: Arrays e Collections

6 Il caso delle java.util.Map

7 Un esercizio sulle collezioni

Interfaccia da implementare

```
1 import java.util.*;
2
3 public interface Graph<N> {
4
5     // Adds a node: nothing happens if node is null or already there
6     void addNode(N node);
7
8     // Adds an edge: nothing happens if source or target are null
9     void addEdge(N source, N target);
10
11    // Returns all the nodes
12    Set<N> nodeSet();
13
14    // Returns all the nodes directly targeted from node
15    Set<N> linkedNodes(N node);
16
17    // Gets one sequence of nodes connecting source to path
18    List<N> getPath(N source, N target);
19
20}
```

Codice di prova

```
1 public class UseGraph{  
2  
3     public static void main(String[] args){  
4         Graph<String> g = null; //new GraphImpl<>();  
5  
6         g.addNode("a");  
7         g.addNode("b");  
8         g.addNode("c");  
9         g.addNode("d");  
10        g.addNode("e");  
11  
12        g.addEdge("a", "b");  
13        g.addEdge("b", "c");  
14        g.addEdge("c", "d");  
15        g.addEdge("d", "e");  
16        g.addEdge("c", "a");  
17  
18        System.out.println(g.nodeSet());  
19        // ["a","b","c","d","e"].. in any order  
20        System.out.println(g.linkedNodes("c"));  
21        // [d,a].. in any order  
22        System.out.println(g.getPath("b", "a"));  
23        // either the path b,c,a or b,c,d,e,a  
24  
25    }  
26  
27 }
```

Strategia risolutiva

Passi:

1. Capire bene cosa la classe deve realizzare
2. Pensare a quale tipo di collezioni può risolvere il problema in modo semplice e prestante
3. Realizzare i vari metodi
4. Controllare i casi particolari