09
Generici
(polimorfismo parametrico)

Mirko Viroli
mirko.viroli@unibo.it

C.D.L. Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2022/2023

Outline

Goal della lezione

- Illustrare il problema delle collezioni polimorfiche
- Discutere il concetto di polimorfismo parametrico
- Illustrare i Generici di Java e alcuni loro vari dettagli

Argomenti

- Collezioni con polimorfismo inclusivo
- Classi generiche
- Interfacce generiche
- Metodi generici

Outline

1. Collections con polimorfismo inclusivo
2. Classi generiche
3. Interfacce generiche
4. Metodi generici
5. Java Wildcards

Forme di riuso nella programmazione OO

Composizione (e come caso particolare, delegazione)

Un oggetto è ottenuto per composizione di oggetti di altre classi

Estensione

Una nuova classe è ottenuta riusando il codice di una classe pre-esistente

Polimorfismo inclusivo (subtyping)

Una funzionalità realizzata per lavorare su valori/oggetti del tipo `A`, può lavorare con qualunque valore/oggetto del sottotipo `B` (p.e., se `B` estende la classe `A`, o se `B` implementa l'interfaccia `A`)

Polimorfismo parametrico (Java/C# generics, C++ templates,..)

Una funzionalità (classe o metodo) generica è costruita in modo tale da lavorare uniformemente su valori/oggetti indipendentemente dal loro tipo: tale tipo diventa quindi una sorta di parametro addizionale

Astrazioni uniformi con le classi

Astrazioni uniformi per problemi ricorrenti

- Durante lo sviluppo di vari sistemi si incontrano problemi ricorrenti che possono trovare una soluzione comune
- In alcuni casi queste soluzioni sono fattorizzabili (per astrazione) in una singola classe altamente riusabile

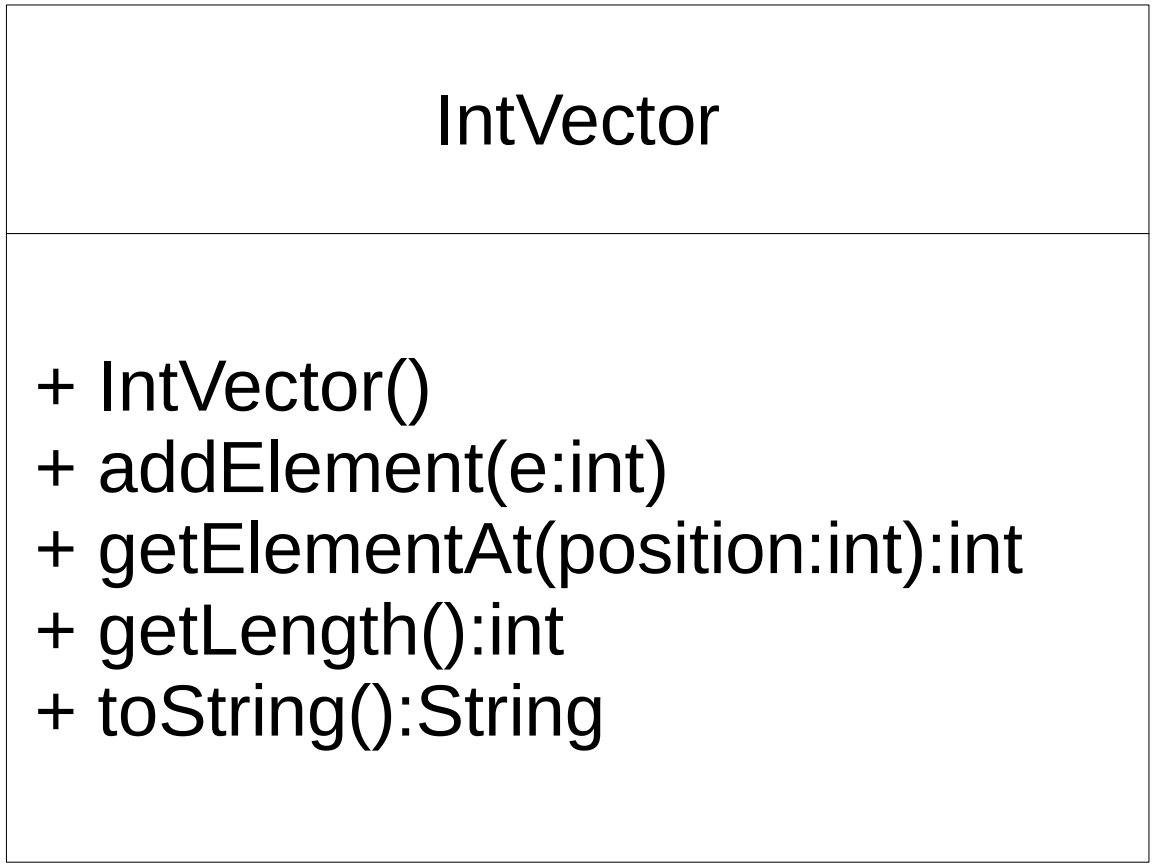
Un caso fondamentale: le collection

- Una collection è un oggetto il cui compito è quello di immagazzinare il riferimento ad un numero (tipicamente non precisato) di altri oggetti
- Fra i suoi compiti c'è quello di consentire modifiche ed accessi veloci all'insieme di elementi di tale collezioni
- Varie strategie possono essere utilizzate, seguendo la teoria/pratica degli algoritmi e delle strutture dati

Un esempio: `IntVector`

Collection `IntVector`

- Contiene serie numeriche (vettori) di dimensione non nota a priori
- Ossia, a lunghezza variabile..



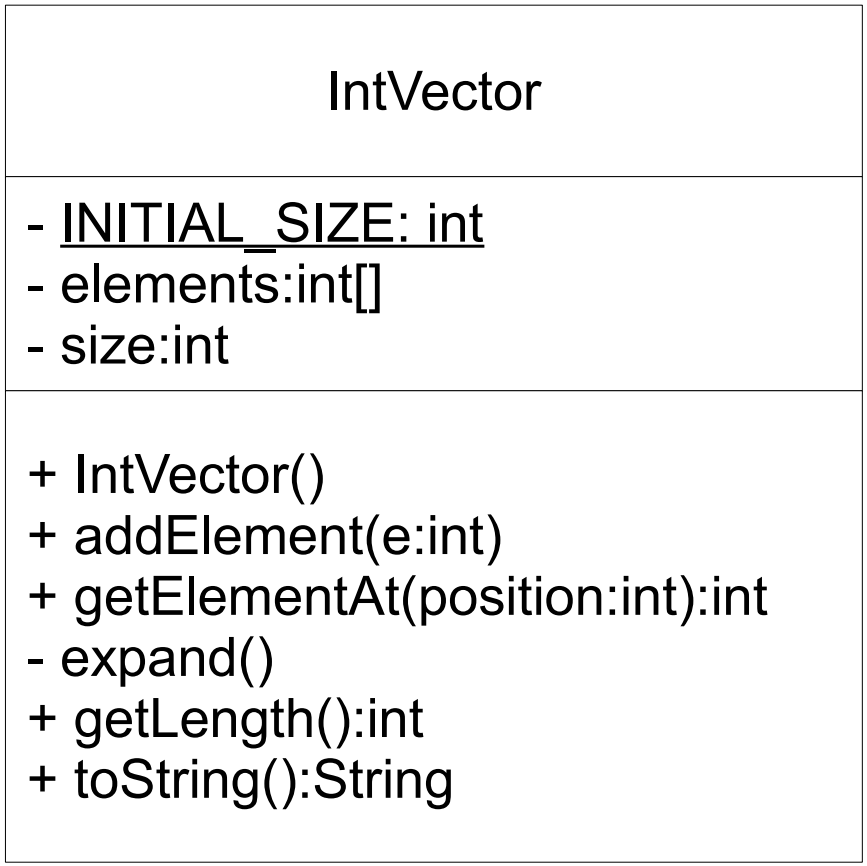
UseIntVector

```java
public class UseIntVector {
  public static void main(String[] s) {
    // Serie di Fibonacci
    final IntVector vi = new IntVector();
    // fib(0)=fib(1)=1, fib(N)=fib(N-1)+fib(N-2) if N>1
    vi.addElement(1);
    vi.addElement(1);
    for (int i = 0; i < 20; i++) {
      vi.addElement(vi.getElementAt(vi.getLength() - 1) + // ultimo
          vi.getElementAt(vi.getLength() - 2) // penultimo
      );
    }
    System.out.println(vi);
    // |1|1|2|3|5|8|13|21|34|55|89|144|233|..
    // 377|610|987|1597|2584|4181|6765|10946|17711|
  }
}
```

IntVector – implementazione

Collection IntVector

- Contiene serie numeriche (vettori) di dimensione non nota a priori
- Realizzata componendo un array che viene espanso all'occorrenza



IntVector pt 1

```java
public class IntVector{

    private static final int INITIAL_SIZE = 10;

    private int[] elements; // Deposito per gli elementi
    private int size;   // Numero di elementi

    public IntVector(){   // Inizialmente vuoto
      this.elements = new int[INITIAL_SIZE];
      this.size = 0;
    }

    public void addElement(final int e){
      if (this.size == elements.length){
          this.expand();  // Se non c'è più spazio..
      }
      this.elements[this.size] = e;
      this.size++;
    }

    public int getElementAt(final int position){
      return this.elements[position];
    }
```

IntVector pt 2

```
1
2       public int getLength(){
3         return this.size;
4       }
5
6       private void expand(){  // Raddoppio lo spazio..
7         final int[] newElements = new int[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        //this.elements = java.util.Arrays.copyOf(this.elements, this
      .elements.length*2);
13      }
14
15      public String toString(){
16        String s="|";
17        for (int i=0; i < size; i++){
18            s = s + this.elements[i] + "|";
19        }
20        return s;
21      }
22  }
```

Un primo passo verso l'uniformità

Solo elenchi di `int`?

- L'esperienza porterebbe subito alla necessità di progettare vettori di `float`, `double`, `boolean`,.. ossia di ogni tipo primitivo
- E poi, anche vettori di `String`, `Date`, eccetera
- L'implementazione sarebbe analoga, ma senza possibilità di riuso..

Collection uniformi "monomorfiche"

- Una prima soluzione del problema la si ottiene sfruttando il polimorfismo inclusivo e la filosofia "everything is an object" (incluso l'uso dell'autoboxing dei tipi primitivi)
- Si realizza unicamente un `ObjectVector`, semplicemente sostituendo `int` con `Object`
- Si inserisce qualunque elemento (via upcast implicito)
- Quando si riottiene un valore serve un downcast esplicito

Da IntVector a ObjectVector

IntVector
- INITIAL_SIZE: int - elements:int[] - size:int
+ IntVector() + addElement(e:int) + getElementAt(position:int):int - expand() + getLength():int + toString():String

ObjectVector
- INITIAL_SIZE: int - elements:Object[] - size:int
+ ObjectVector() + addElement(e:Object) + getElementAt(position:int):Object - expand() + getLength():int + toString():String

UseObjectVector

```java
public class UseObjectVector {
  public static void main(String[] s) {
    // Serie di Fibonacci
    final ObjectVector vobj = new ObjectVector();
    // fib(0)=fib(1)=1, fib(N)=fib(N-1)+fib(N-2) if N>1
    vobj.addElement(1); // grazie all'autoboxing
    vobj.addElement(1);
    for (int i = 0; i < 20; i++) {
      vobj.addElement( // servono downcast specifici
      (Integer) vobj.getElementAt(vobj.getLength() - 1)
          + (Integer) vobj.getElementAt(vobj.getLength() - 2));
    }
    System.out.println(vobj);
    // |1|1|2|3|5|8|13|21|34|55|89|144|233|..
    // 377|610|987|1597|2584|4181|6765|10946|17711|

    // Altro esempio
    final ObjectVector vobj2 = new ObjectVector();
    vobj2.addElement("Prova");
    vobj2.addElement("di");
    vobj2.addElement("vettore");
    vobj2.addElement(new Object());
    System.out.println(vobj2);
    String str = (String) vobj2.getElementAt(1); // "di"
    // String str2 = (String)vobj2.getElementAt(3); // Exception
  }
}
```

Un altro caso di collection, la list linkata ObjectList

```java
/* Lista linkata di oggetti, con soli metodi getter */
public class ObjectList {

  private final Object head;
  private final ObjectList tail;

  public ObjectList(final Object head, final ObjectList tail) {
    this.head = head;
    this.tail = tail;
  }

  public Object getHead() { // Testa della lista
    return this.head;
  }

  public ObjectList getTail() { // Coda della lista
    return this.tail;
  }

  public int getLength() { // Dimensione della lista
    return (this.tail == null) ? 1 : 1 + this.tail.getLength();
  }

  public String toString() { // Rappr. a stringa
    return "|" + this.head
        + ((this.tail == null) ? "|" : this.tail.toString());
  }
}
```

UseObjectList

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

La necessità di un approccio a polimorfismo parametrico

Prima di Java 5

- Questo era l'approccio standard alla costruzione di collection
- Java Collection Framework — una libreria fondamentale

Problema

- Con questo approccio, nel codice Java risultavano molti usi di oggetti simili a `ObjectVector` o `ObjectList`
- Si perdeva molto facilmente traccia di quale fosse il contenuto..
  - contenevano oggetti vari? solo degli `Integer`? solo delle `String`?
- Il codice conteneva spesso dei downcast sbagliati, e quindi molte applicazioni Java fallivano generando dei `ClassCastException`

Più in generale

Il problema si manifesta ogni volta che voglio collezionare oggetti il cui tipo non è noto a priori, ma potrebbe essere soggetto a polimorfismo inclusivo

Polimorfismo parametrico

Idea di base

- Dato un frammento di codice `F` che lavora su un certo tipo, diciamo `String`, se potrebbe anche lavorare in modo uniforme con altri..
- ..lo si rende parametrico, sostituendo a `String` una sorta di variabile `X` (chiamata type-variable, ossia una variabile che contiene un tipo)
- A questo punto, quando serve il frammento di codice istanziato sulle stringhe, si usa `F<String>`, ossia si richiede che `X` diventi `String`
- Quando serve il frammento di codice istanziato sugli integer, si usa `F<Integer>`

Java Generics

- Classi/interfacce/metodi generici
- Nessun impatto a run-time, per via dell'implementazione a "erasure"
  - `javac` "compila via i generici", quindi la JVM non li vede

Outline

1. Collections con polimorfismo inclusivo
2. Classi generiche
3. Interfacce generiche
4. Metodi generici
5. Java Wildcards

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        final List<Integer> list =
4            new List<Integer>(10, // Autoboxing
5              new List<Integer>(20,
6              new List<Integer>(30,
7              new List<Integer>(40,null))));
8        // Cast NON necessari
9        final int first = list.getHead(); // Unboxing
10       final int second = list.getTail().getHead();
11       final int third = list.getTail().getTail().getHead();
12       System.out.println(first+" "+second+" "+third);
13       System.out.println(list.toString());
14       System.out.println(list.getLength());
15
16       // Usabile anche con le stringhe
17       final List<String> list2 = new List<String>("a",
18                     new List<String>("b",
19                     new List<String>("c",
20                     new List<String>("d",null))));
21       System.out.println(list2.toString());
22     }
23 }
```

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 usati 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

La classe generica Vector

```
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.getElementAt(0) + " " +
11               vs.getElementAt(1) + " " +
12               vs.getElementAt(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.getElementAt(0) + // Unboxing
20         vi.getElementAt(1) +
21         vi.getElementAt(2);
22       System.out.println(i);
23     }
24 }
```

Implementazione di Vector pt 1

```
1  public class Vector<X>{
2
3      private final static int INITIAL_SIZE = 10;
4
5      private Object[] elements;  // No X[], devo usare Object[]!!
6      private int size;
7
8      public Vector(){
9        this.elements = new Object[INITIAL_SIZE]; // Object[]
10       this.size = 0;
11     }
12
13     public void addElement(X e){  // Tutto come atteso
14       if (this.size == elements.length){
15           this.expand();
16       }
17       this.elements[this.size] = e;
18       this.size++;
19     }
20     ...
```

Implementazione di Vector pt 2

```
1     ...
2     public X getElementAt(int position){
3       return (X)this.elements[position]; // Conversione a X
4       // Genera un unchecked warning!
5     }
6
7     private void expand(){
8       // Ancora Object[]
9       Object[] newElements = new Object[this.elements.length*2];
10      for (int i=0; i < this.elements.length; i++){
11          newElements[i] = this.elements[i];
12      }
13      this.elements = newElements;
14    }
15
16    // getLength() e toString() inalterate
17    ...
18 }
```

La classe generica Pair<X,Y>

```
1  public class Pair<X, Y> {
2
3    private final X first;
4    private final Y second;
5
6    public Pair(final X first, final Y second) {
7      this.first = first;
8      this.second = second;
9    }
10
11   public X getFirst() {
12     return this.first;
13   }
14
15   public Y getSecond() {
16     return this.second;
17   }
18
19   public String toString() {
20     return "<" + this.first + "," + this.second + ">";
21   }
22 }
```

Uso di Pair<X,Y>

```
1  public class UsePair{
2      public static void main(String[] s){
3        Pair<String,Integer> p = new Pair<String,Integer>("aa",1);
4        String fst = p.getFirst();
5        int snd = p.getSecond();
6        System.out.println(fst+" "+snd);
7
8        final Vector<Pair<String,Integer>> v =
9            new Vector<Pair<String,Integer>>();
10       v.addElement(new Pair<String,Integer>("Prova",1));
11       v.addElement(new Pair<String,Integer>("di",2));
12       v.addElement(new Pair<String,Integer>("Vettore",3));
13       final String str = v.getElementAt(0).getFirst() + " " +
14              v.getElementAt(1).getFirst() + " " +
15              v.getElementAt(2).getFirst(); // Nota, nessun cast!
16       System.out.println(str);
17       System.out.println(v);
18
19       final List<Pair<Integer,Integer>> l =
20           new List<Pair<Integer,Integer>>(new Pair<Integer,Integer>(1,1),
21           new List<Pair<Integer,Integer>>(new Pair<Integer,Integer>(2,2),
22           new List<Pair<Integer,Integer>>(new Pair<Integer,Integer>(3,3),
23           null)));
24       System.out.println(l);
25     }
26 }
```

Inferenza dei parametri

Un problema sintattico dei generici

- Tendono a rendere il codice più pesante ("verbose")
- Obbligano a scrivere i parametri anche dove ovvi, con ripetizioni

L'algoritmo di type-inference nel compilatore

- Nella `new` si possono tentare di omettere i parametri (istanziazione delle type-variable), indicando il "diamond symbol" <>
- Il compilatore cerca di capire quali siano questi parametri guardando gli argomenti della `new` e l'eventuale contesto dentro il quale la `new` è posizionata, per esempio, se assegnata ad una variabile
- Nel raro caso in cui non ci riuscisse, segnalerebbe un errore a tempo di compilazione.. quindi tanto vale provare!
- Ricordarsi <>, altrimenti viene confuso con un raw type, un meccanismo usato per gestire il legacy con le versioni precedenti di Java

La local variable type inference

- in genere è alternativa al simbolo <>

Esempi di inferenza

```
1  public class UsePair2{
2      public static void main(String[] s){
3        // Parametri in new Vector() inferiti dal tipo della variabile
4        final Vector<Pair<String,Integer>> v = new Vector<>();
5        // Parametri in new Pair(..) inferiti dal tipo degli argomenti
6        v.addElement(new Pair<>("Prova",1));
7        v.addElement(new Pair<>("di",2));
8        v.addElement(new Pair<>("Vettore",3));
9        final String str = v.getElementAt(0).getFirst() + " " +
10               v.getElementAt(1).getFirst() + " " +
11               v.getElementAt(2).getFirst(); // Nota, nessun cast!
12       System.out.println(str);
13       System.out.println(v);
14
15         // Inferenza grazie agli argomenti e tipo variabile..
16       final List<Pair<Integer,Integer>> l =
17           new List<>(new Pair<>(1,1),
18           new List<>(new Pair<>(2,2),
19           new List<>(new Pair<>(3,3),
20           null)));
21       System.out.println(l);
22
23       // Local variable type inference
24       final var v2 = new Vector<Integer>();
25       v2.addElement(1);
26       System.out.println(v2);
27     }
28 }
```

I vantaggi dei generici

Coi generici, Java diventa un linguaggio molto più espressivo!

Svantaggi

- Il linguaggio risulta più sofisticato, e quindi complesso
- Se non ben usati, possono minare la comprensibilità del software
- Non vanno abusati!!
- Gli aspetti più avanzati dei generici, che vedremo, sono considerati troppo complessi

Vantaggi – se ben usati

- Codice più comprensibile
- Maggiori possibilità di riuso di funzionalità (quasi d'obbligo oramai)
- Codice più sicuro (safe) – il compilatore segnala errori difficili da trovare altrimenti

Outline

1. Collections con polimorfismo inclusivo
2. Classi generiche
3. Interfacce generiche
4. Metodi generici
5. Java Wildcards

Interfacce generiche

Cosa è una interfaccia generica

- È una interfaccia che dichiara type-variables:
  `interface I<X,Y>{..}`
- Le type-variable compaiono nei metodi definiti dall'interfaccia
- Quando una classe la implementa deve istanziare le type variabile (o assegnarle ad altre type-variable se essa stessa è generica)

Utilizzi

Per creare contratti uniformi che non devono dipendere dai tipi utilizzati

Un esempio notevole, gli Iteratori

- Un iteratore è un oggetto usato per accedere ad una sequenza di elementi
- Ne vedremo ora una versione semplificata – diversa da quella delle librerie Java

L'interfaccia Iterator

```java
public interface Iterator<E> {

  // torna il prossimo elemento dell'iterazione
  E next();

  // dice se vi saranno altri elementi
  boolean hasNext();

  /*
   * Nota: non è noto cosa succede se si chiama
   * next() quando hasNext() ha dato esito falso
   */
}
```

Implementazione 1: IntRangeIterator

```
1  /* Itera tutti i numeri interi fra 'start' e 'stop' inclusi */
2  public class IntRangeIterator implements Iterator<Integer> {
3
4    private int current; // valore corrente
5    private final int stop; // valore finale
6
7    public IntRangeIterator(final int start, final int stop) {
8      this.current = start;
9      this.stop = stop;
10   }
11
12   public Integer next() {
13     return this.current++;
14   }
15
16   public boolean hasNext() {
17     return this.current <= this.stop;
18   }
19 }
```

Implementazione 2: ListIterator

```
1  /* Itera tutti gli elementi di una List */
2  public class ListIterator<E> implements Iterator<E> {
3
4    private List<E> list; // Lista corrente
5
6    public ListIterator(final List<E> list) {
7      this.list = list;
8    }
9
10   public E next() {
11     final E element = this.list.getHead();// Elemento da tornare
12     this.list = this.list.getTail();// Aggiorno la lista
13     return element;
14   }
15
16   public boolean hasNext() {
17     return (this.list != null);
18   }
19 }
```

Implementazione 3: VectorIterator

```java
/* Itera tutti gli elementi di un Vector */
public class VectorIterator<E> implements Iterator<E> {

  private final Vector<E> vector; // Vettore da iterare
  private int current; // Posizione nel vettore

  public VectorIterator(final Vector<E> vector) {
    this.vector = vector;
    this.current = 0;
  }

  public E next() {
    return this.vector.getElementAt(this.current++);
  }

  public boolean hasNext() {
    return this.vector.getLength() > this.current;
  }
}
```

UseIterators: nota l'accesso uniforme!

```
1  public class UseIterators {
2    public static void main(String[] s) {
3      final List<Integer> list = new List<>(1,
4                              new List<>(2,
5                                new List<>(3, null)));
6
7      final Vector<Integer> vector = new Vector<>();
8      vector.addElement(10);
9      vector.addElement(20);
10
11     // creo 3 iteratori..
12     final Iterator<Integer> iterator = new IntRangeIterator(5, 10);
13     final Iterator<Integer> iterator2 = new ListIterator<>(list);
14     final Iterator<Integer> iterator3 = new VectorIterator<>(vector);
15
16     // ne stampo il contenuto..
17     printAll(iterator);
18     printAll(iterator2);
19     printAll(iterator3);
20   }
21
22   // e se volessi una printAll che prende anche Iterator<String>??
23   static void printAll(Iterator<Integer> iterator) {
24     while (iterator.hasNext()) {
25       System.out.println("Elemento : " + iterator.next());
26     }
27   }
28 }
```

Outline

1. Collections con polimorfismo inclusivo
2. Classi generiche
3. Interfacce generiche
4. Metodi generici
5. Java Wildcards

Metodi generici

Metodo generico

Un metodo che lavora su qualche argomento e/o valore di ritorno in modo independente dal suo tipo effettivo. Tale tipo viene quindi astratto in una type-variable del metodo.

Sintassi

- def: `<X1,..,Xn> ret-type nome-metodo(formal-args){...}`
- call: `receiver.<X1,..,Xn>nome-metodo(actual-args){...}`
- call con inferenza, stessa sintassi delle call standard, ossia senza <>

Due casi principali, con medesima gestione

- Metodo generico (statico o non-statico) in una classe non generica
- Metodo generico (non-statico) in una classe generica

$\Rightarrow$ Il primo dei due molto più comune..

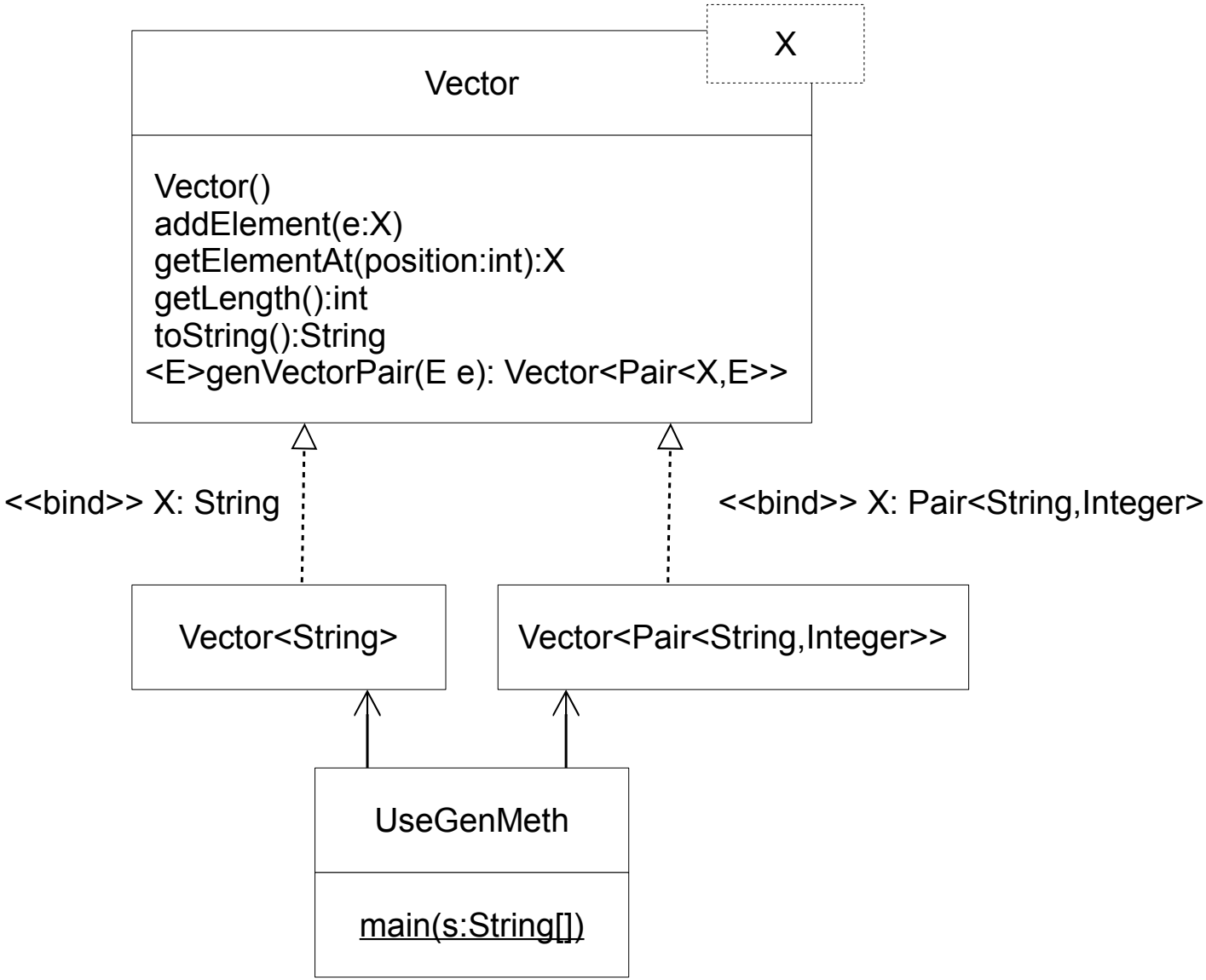
UseIterators: Definizione e uso di metodo generico

```
1  public class UseIterators2{
2      public static void main(String[] s){
3        final Iterator<Integer> iterator = new IntRangeIterator(5,10);
4
5        final List<String> list = new List<String>("a",
6                    new List<String>("b",
7                    new List<String>("c",null)));
8        final Iterator<String> iterator2 = new ListIterator<>(list);
9
10       final Vector<java.util.Date> vector=new Vector<>(); // inferenza
11       vector.addElement(new java.util.Date());
12       vector.addElement(new java.util.Date());
13
14         final Iterator<java.util.Date> iterator3 = new VectorIterator<>(
       vector);
15
16         UseIterators2.<Integer>printAll(iterator);
17         UseIterators2.<String>printAll(iterator2);
18         UseIterators2.<java.util.Date>printAll(iterator3);
19
20         UseIterators2.printAll(iterator3); // invocazione con inferenza
21     }
22     // metodo generico
23     public static <E> void printAll(final Iterator<E> iterator){
24       while (iterator.hasNext()){
25           System.out.println("Elemento : "+iterator.next());
26       }
27     }
```

Esempio di metodo generico in classe generica

```
1  public class Vector<X>{
2
3      ...
4      <E> Vector<Pair<X,E>> genVectorPair(E e){
5        Vector<Pair<X,E>> vp = new Vector<>(); /// Inferenza
6        for (int i=0;i<this.size;i++){
7            vp.addElement(new Pair<>(this.getElementAt(i),e));
8        }
9        return vp;
10     }
11 }
12
13 public class UseGenMeth{
14     public static void main(String[] s){
15       Vector<String> vs = new Vector<>();
16       vs.addElement("prova");
17       vs.addElement("di");
18       vs.addElement("vettore");
19       Vector<Pair<String,Integer>> vp = vs.<Integer>genVectorPair(101);
20       // versione con inferenza..
21       // Vector<Pair<String,Integer>> vp2 = vs.genVectorPair(101);
22       System.out.println(vp);
23       // |<prova,101>|<di,101>|<vettore,101>|
24     }
25 }
```

Notazione UML: non del tutto standard



Outline

1. Collections con polimorfismo inclusivo
2. Classi generiche
3. Interfacce generiche
4. Metodi generici
5. Java Wildcards

Java Wildcards

Osservazione

- Esistono situazioni in cui un metodo debba accettare come argomento non solo oggetti di un tipo `C<T>`, ma di ogni `C<S>` dove `S <: T`
- Esempio: un metodo statico `printAll()` che prende in ingresso un iteratore e ne stampa gli elementi
- È realizzabile con un metodo generico, ma ci sono casi in cui si vorrebbe esprimere un tipo che raccolga istanziazioni diverse di una classe generica

Java Wildcards

- Un meccanismo avanzato, quello inventato più di recente (2004-2006)
  - (Igarashi & Viroli) + (Bracha & Gafter) + (Torgersen & Hansen & von der Ahé)
- Fornisce dei nuovi tipi, chiamati Wildcards
- Simili a interfacce (non generano oggetti, descrivono solo contratti)
- Generalmente usati come tipo dell'argomento di metodi

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

- 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

Esempio Wildcard

```
1  public class Wildcard{
2
3    // Metodo che usa la wildcard
4      public static void printAll(Iterator<?> it){
5        while (it.hasNext()){
6          System.out.println(it.next());
7        }
8      }
9
10   // Analoga versione con metodo generico
11     public static <T> void printAll2(Iterator<T> it){
12       while (it.hasNext()){
13         System.out.println(it.next());
14       }
15     }
16
17     // Quale versione preferibile?
18
19     public static void main(String[] s){
20       Wildcard.printAll(new IntRangeIterator(1,5));
21       Wildcard.printAll2(new IntRangeIterator(1,5));
22       Wildcard.<Integer>printAll2(new IntRangeIterator(1,5));
23     }
24 }
```