Capitolo 16

Programmazione generica

Cay S. Horstmann
Concetti di informatica e fondamenti di Java
quarta edizione

Obiettivi del capitolo

- Capire gli obiettivi della programmazione generica
- Essere in grado di realizzare classi e metodi generici
- Comprendere il meccanismo di esecuzione di metodi generici all'interno della macchina virtuale Java
- Conoscere le limitazioni relative alla programmazione generica in Java
- Capire la relazione esistente tra tipi generici ed ereditarietà
- Imparare a porre vincoli sulle variabili di tipo

Variabili di tipo

- Programmazione generica: creazione di costrutti di programmazione che possano essere utilizzati con tipi di dati diversi. In Java si può raggiungere l'obiettivo della programmazione generica usando l'ereditarietà oppure le variabili di tipo.
- Esempio:
  Variabile di tipo:*`ArrayList` di Java (e.g. `ArrayList<String`>)*
  Ereditarietà*: `LinkedList`* che abbiamo realizzato nel paragrafo 14.2 può memorizzare oggetti di qualsiasi tipo
- Classe generica: è stata dichiarata usando una variabile di tipo E. La variabile di tipo rappresenta il tipo degli elementi:

```
public class ArrayList<E>
         // si può usare "ElementType" invece di E
{
   public ArrayList() { . . . }
   public void add(E element) { . . . }
   . . .
}
```

Variabili di tipo

- Le variabili di tipo possono essere sostituite, all'atto della creazione di esemplari, con nomi di classe o di interfacce

```
ArrayList<BankAccount>
ArrayList<Measurable>
```

- Non si può usare come sostituto uno degli otto tipi di dati primitivi

```
ArrayList<double> // Sbagliato!
```

- Usate un esemplare di `ArrayList<Double>`

Variabili di tipo

- Il tipo di dato che indicate va a sostituire la variabile di tipo utilizzata nella definizione dell'interfaccia o della classe generica
- Esempio: nel metodo `add` di un oggetto di tipo `ArrayList<BankAccount>` la variabile di tipo E viene sostituita dal tipo `BankAccount`

```
public void add(BankAccount element)
```

- Cofrontate invece quanto accade nel metodo `add` della classe `LinkedList`

```
public void add(Object element)
```

Variabili di tipo

- Le variabili di tipo rendono più sicuro e di più facile comprensione il codice generico.

- E' impossibile aggiungere un oggetto di tipo `String` ad un esemplare di `ArrayList<BankAccount>` mentre è possibile aggiungere un oggetto di tipo `String` ad un esemplare di `LinkedList` che sia stato creato con l'intenzione di usarlo per contenere conti correnti

```
ArrayList<BankAccount> accounts1 = new ArrayList<BankAccount>();
LinkedList accounts2 = new LinkedList();
      // Dovrebbe contenere oggetti di tipo BankAccount
accounts1.add("my savings"); // errore durante la compilazione
accounts2.add("my savings");
      // errore non individuato dal compilatore
. . .
BankAccount account = (BankAccount) accounts2.getFirst();
// errore durante l'esecuzione
```

Sintassi di Java 16.1: Definizione di un tipo specifico di classe generica

NomeClasseGenerica<Tipo1, Tipo2, . . .>

Esempio:

```
ArrayList<BankAccount>
HashMap<String, Integer>
```

Obiettivo:

Fornire i tipi specifici che vadano a sostituire le variabili di tipo presenti in una classe generica

Realizzare classi generiche

- Esempio: una classe generica semplice che memorizza coppie di oggetti

```
 Pair<String, BankAccount> result
= new Pair<String, BankAccount>("Harry Hacker",
harrysChecking);
```

- I metodi `getFirst` e `getSecond` restituiscono il primo e il secondo valore memorizzati nella coppia

```
 String name = result.getFirst();
 BankAccount account = result.getSecond();
```

- Questa classe può essere utile quando si realizza un metodo che calcola e deve restituire due valori: un metodo non può restituire contemporaneamente un esemplare di `String` e un esemplare di `BankAccount` mentre può restituire un singolo oggetto di tipo `Pair<String,BankAccount>`.

- La classe generica `Pair` richiede due variabili di tipo, una per il tipo del primo elemento e una per il tipo del secondo elemento

Variabili di tipo

Nome della variabile di tipo	Significato
E	Tipo di un elemento in una raccolta
K	Tipo di una chiave in una mappa
V	Tipo di un valore in una mappa
T	Tipo generico
S, U	Ulteriori tipi generici

La classe `Pair`

```
public class Pair<T, S>
{
   public Pair(T firstElement, S secondElement)
   {
      first = firstElement;
      second = secondElement;
   }
   public T getFirst() { return first; }
   public S getSecond() { return second; }

   private T first;
   private S second;
}
```

Trasformare in classe generica la classe `LinkedList`

```
public class LinkedList<E>
{
   . . .
   public E removeFirst()
   {
      if (first == null)
         throw new NoSuchElementException();
      E element = first.data;
      first = first.next;
      return element;
   }
   private Node first;

   private class Node
   {
      public E data;
      public Node next;
   }
}
```

Realizzare classi generiche

- Usate il tipo E ogni volta che ricevete, restituite o memorizzate un oggetto che sia un dato della lista concatenata, un “elemento”

- Le cose si complicano un po’ soltanto quando la vostra struttura dati usa classi ausiliarie

- Se si tratta di classi interne non dovete fare nulla di speciale

- Se invece tali classi ausiliarie vengono definite al di fuori della classe generica, devono anch’esse diventare classi generiche

```
public class ListNode<E>
```

Sintassi di Java 16.2 Definizione di una classe generica

```
specificatoreDiAccesso class
NomeClasseGenerica<VariabileDiTipo1, VariabileDiTipo2, . . .>

{
   costruttori
   metodi
   campi
}
```

Esempio:

```
public class Pair<T, S>
{
   . . .
}
```

Obiettivo:

Definire una classe generica con metodi e campi che dipendono da variabili di tipo

File LinkedList.java (continua)

```
001: import java.util.NoSuchElementException;
002:
003: /**
004:    Una lista concatenata è una sequenza di nodi dotata di
005:    efficienti meccanismi di inserimento e eliminazione.
006:    Questa classe ha un sottoinsieme dei metodi della classe
007:    standard java.util.LinkedList.
008: */
009: public class LinkedList<E>
010: {
011:    /**
012:       Costruisce una lista concatenata vuota.
013:    */
014:    public LinkedList()
015:    {
016:       first = null;
017:    }
018:
019:    /**
020:       Restituisce il primo elemento della lista concatenata.
021:       @return il primo elemento della lista concatenata
022:    */
```

File LinkedList.java (continua)

```
023:    public E getFirst()
024:    {
025:       if (first == null)
026:          throw new NoSuchElementException();
027:       return first.data;
028:    }
029:
030:    /**
031:       toglie il primo elemento dalla lista concatenata.
032:       @return l'elemento eliminato
033:    */
034:    public E removeFirst()
035:    {
036:       if (first == null)
037:          throw new NoSuchElementException();
038:       E element = first.data;
039:       first = first.next;
040:       return element;
041:    }
042:
043:    /**
044:       Aggiunge un elemento in testa alla lista concatenata.
045:       @param element l'elemento da aggiungere
046:    */
```

File LinkedList.java(continua)

```
047:    public void addFirst(E element)
048:    {
049:       Node newNode = new Node();
050:       newNode.data = element;
051:       newNode.next = first;
052:       first = newNode;
053:    }
054:
055:    /**
056:       Restituisce un iteratore per questa lista.
057:       @return un iteratore per questa lista
058:    */
059:    public ListIterator<E> listIterator()
060:    {
061:       return new LinkedListIterator();
062:    }
063:
064:    private Node first;
065:
066:    private class Node
067:    {
```

File LinkedList.java (continua)

```
068:        public E data;
069:        public Node next;
070:     }
071:
072:     private class LinkedListIterator implements ListIterator<E>
073:     {
074:        /**
075:           Costruisce un iteratore che punta alla testa
076:           della lista concatenata.
077:        */
078:        public LinkedListIterator()
079:        {
080:           position = null;
081:           previous = null;
082:        }
083:
084:        /**
085:           Sposta l'iteratore dopo l'elemento successivo.
086:           @return l'elemento attraversato
087:        */
088:        public E next()
089:        {
```

File LinkedList.java (continua)

```
090:          if (!hasNext())
091:             throw new NoSuchElementException();
092:          previous = position; // memorizza per remove
093:
094:          if (position == null)
095:             position = first;
096:          else
097:             position = position.next;
098:
099:          return position.data;
100:       }
101:
102:       /**
103:          Controlla se c'è un elemento dopo la posizione
104:           dell'iteratore
105:          @return true se e solo se c'è un elemento dopo la
106:          posizione dell'iteratore
107:       */
108:       public boolean hasNext()
109:       {
110:          if (position == null)
111:             return first != null;
```

File LinkedList.java (continua)

```
112:          else
113:             return position.next != null;
114:       }
115:
116:       /**
117:          Aggiunge un elemento prima della posizione dell'iteratore
118:          e sposta l'iteratore dopo l'elemento inserito.
119:          @param element l'oggetto da aggiungere
120:       */
121:       public void add(E element)
122:       {
123:          if (position == null)
124:          {
125:             addFirst(element);
126:             position = first;
127:          }
128:          else
129:          {
130:             Node newNode = new Node();
131:             newNode.data = element;
132:             newNode.next = position.next;
133:             position.next = newNode;
```

File LinkedList.java (continua)

```
134:             position = newNode;
135:          }
136:          previous = position;
137:       }
138:
139:       /**
140:        Toglie l'ultimo elemento attrversato. Questo metodo può
141:       essere chiamato soltanto dopo un'invocazione del metodo next()
142:        */
143:       public void remove()
144:       {
145:          if (previous == position)
146:             throw new IllegalStateException();
147:
148:          if (position == first)
149:          {
150:             removeFirst();
151:          }
152:          else
153:          {
154:             previous.next = position.next;
155:          }
```

File LinkedList.java

```
156:            position = previous;
157:         }
158:
159:         /**
160:            Imposta il dato presente nell'ultimo elemento
161:             attraversato
162:            @param element il dato da impostare
163:         */
164:         public void set(E element)
165:         {
166:            if (position == null)
167:               throw new NoSuchElementException();
168:            position.data = element;
169:         }
170:
171:         private Node position;
172:         private Node previous;
173:      }
174: }
```

File ListIterator.java (continua)

```
01: /**
02:    Un iteratore di lista consente l'accesso a una posizione in una

03:    lista concatenata. Questa interfaccia contiene un sottoinsieme
04:    dei metodi dell'interfaccia standard java.util.ListIterator,
05:    in quanto mancano i metodi per l'attraversamento all'indietro.
06: */
07: public interface ListIterator<E>
08: {
09:    /**
10:       Sposta l'iteratore dopo l'elemento successivo.
11:       @return l'elemento attraversato
12:    */
13:    E next();
14:
15:    /**
16:       Controlla se c'è un elemento dopo la posizione
17:       dell'iteratore.
18:       @return true se e solo se c'è un elemento dopo la posizione
19:        dell'iteratore
20:    */
21:    boolean hasNext();
```

File ListIterator.java

```
22:
23:    /**
24:       Aggiunge un elemento prima della posizione dell'iteratore
25:       e sposta l'iteratore dopo l'elemento inserito.
26:       @param element l'oggetto da aggiungere
27:    */
28:    void add(E element);
29:
30:    /**
31:      Toglie l'ultimo elemento attraversato. Questo metodo può
32:      essere chiamato soltanto dopo un'invocazione del metodo next()
33:    */
34:    void remove();
35:
36:    /**
37:       Imposta il dato presente nell'ultimo elemento attraversato.
38:       @param element il dato da impostare
39:    */
40:    void set(E element);
41: }
```

File ListTester.java (continua)

```
01: /**
02:    Programma che collauda la classe LinkedList
03: */
04: public class ListTester
05: {
06:    public static void main(String[] args)
07:    {
08:       LinkedList<String> staff = new LinkedList<String>();
09:       staff.addFirst("Tom");
10:       staff.addFirst("Romeo");
11:       staff.addFirst("Harry");
12:       staff.addFirst("Dick");
13:
14:       // il segno | nei commenti indica la posizione dell'iteratore
15:
16:       ListIterator<String> iterator = staff.listIterator(); // |DHRT
17:       iterator.next(); // D|HRT
18:       iterator.next(); // DH|RT
19:
20:       // Aggiunge altri elementi dopo il secondo
21:
```

File ListTester.java (continua)

```
22:         iterator.add("Juliet"); // DHJ|RT
23:         iterator.add("Nina"); // DHJN|RT
24:
25:         iterator.next(); // DHJNR|T
26:
27:         // Toglie l'ultimo elemento attraversato
28:
29:         iterator.remove(); // DHJN|T
30:
31:         // stampa tutti gli elementi
32:
33:         iterator = staff.listIterator();
34:         while (iterator.hasNext())
35:         {
36:            String element = iterator.next();
37:            System.out.print(element + " ");
38:         }
39:         System.out.println();
40:         System.out.println("Expected: Dick Harry Juliet Nina Tom");
41:     }
42: }
```

File ListTester.java

Visualizza:

```
Dick Harry Juliet Nina Tom
Expected: Dick Harry Juliet Nina
Tom
```

Metodi generici

- Un metodo generico è un metodo con una variabile di tipo
- Lo si può definire all'interno di classi normali o generiche
- Un metodo che opera su un tipo specifico:

```
/**
   Stampa tutti gli elementi contenuti in un array di stringhe.

   @param a l'array da stampare
*/
public static void print(String[] a)
{
   for (String e : a)
      System.out.print(e + " ");
   System.out.println();
}
```

Metodi generici

- E se invece ci interessasse stampare un array di oggetti di tipo `Rectangle`?

```
public static <E> void print(E[] a)
{
   for (E e : a)
      System.out.print(e + " ");
   System.out.println();
}
```

Metodi generici

- Quando invocate il metodo generico non dovete specificare il tipo effettivo da usare al posto delle variabili di tipo

```
Rectangle[] rectangles = . . .;
ArrayUtil.print(rectangles);
```

- Il compilatore deduce che il tipo effettivo da usare per E è `Rectangle`
- Potete definire anche metodi generici che non siano statici
- Potete infine definire metodi generici all'interno di classi generiche
- Non potete usare tipi primitivi per sostituire variabili di tipo. Ad esempio, non si può usare il metodo `print` per stampare un array di tipo `int[]`

Sintassi di Java 16.3 Definizione di un metodo generico

```
modificatori <VariabileDiTipo1, VariabileDiTipo2, . . .>
tipoRestituito nomeMetodo(parametri)
{
   corpo
}
```

Esempio:

```
public static <E> void print( E[] a)
{
   . . .
}
```

Obiettivo:

Definire un metodo generico che dipenda da variabili di tipo

Vincolare le variabili di tipo

- Le variabili di tipo possono essere vincolate mediante opportuni limiti

```
public static <E extends Comparable> E min(E[] a)
{
   E smallest = a[0];
   for (int i = 1; i < a.length; i++)
      if (a[i].compareTo(smallest) < 0) smallest = a[i];
   return smallest;
}
```

- Potete invocare `min` con un array di tipo `String []` ma non con un array di tipo `Rectangle []` : la classe `String` realizza `Comparable`, ma `Rectangle` no.

- La limitazione `Comparable` è necessaria per poter invocare il metodo `CompareTo` altrimenti il metodo `min` non sarebbe compilato

Vincolare le variabili di tipo

- Raramente vi capiterà di dover indicare due o più vincoli

```
<E extends Comparable & Cloneable>
```

- La parola chiave `extends` quando viene applicata alle variabili di tipo, significa in realtà “estende o realizza”
- I vincoli possono essere classi o interfacce
- Le variabili di tipo possono essere sostituite con il tipo di una classe o di un’interfaccia

Tipi con carattere jolly (*wildcard*)

Nome	Sintassi	Significato
Vincolo con limite inferiore	? extends B	Qualsiasi sottotipo di B
Vincolo con limite superiore	? super B	Qualsiasi supertipo di B
Nessun vincolo	?	Qualsiasi tipo

Tipi con carattere jolly (*wildcard*)

```
public void addAll(LinkedList<? extends E> other)
{
   ListIterator<E> iter = other.listIterator();
   while (iter.hasNext()) add(iter.next());
}

public static <E extends Comparable<E>> E min(E[] a)

public static <E extends Comparable<?super E>>
       E min(E[] a)

public static void reverse(List<?> list)
```

Una dichiarazione di questo tipo è un'abbreviazione per:

```
public static <T> void reverse(List<T> list)
```

Tipi “grezzi” (*raw*)

- La macchina virtuale lavora con tipi “grezzi” (raw), non con classi generiche
- Il tipo grezzo corrispondente a un tipo generico si ottiene eliminando le variabili di tipo
- Ad esempio, la classe generica `Pair<T, S>` viene sostituita dalla seguente classe grezza:

```
public class Pair
{
   public Pair(Object firstElement, Object secondElement)
   {
      first = firstElement;
      second = secondElement;
   }
   public Object getFirst() { return first; }
   public Object getSecond() { return second; }

   private Object first;
   private Object second;
}
```

Tipi “grezzi” (*raw*)

- Ai metodi generici viene applicato lo stesso procedimento

```
public static Comparable min(Comparable[] a)
{
   Comparable smallest = a[0];
   for (int i = 1; i < a.length; i++)
      if (a[i].compareTo(smallest) < 0) smallest = a[i];
   return smallest;
}
```

- Conoscere l’esistenza dei tipi grezzi aiuta a capire i limiti della programmazione generica in Java.

- Per esempio, non potete sostituire le variabili di tipo con i tipi primitivi

- Per utilizzare il codice “vecchio” (legacy) potete effettuare la conversione tra tipi generici e tipi grezzi