16

Programmazione generica

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

La programmazione generica riguarda la progettazione e realizzazione di strutture di dati e di algoritmi che siano in grado di funzionare con tipi di dati diversi. Conoscete già la classe generica `ArrayList`, i cui esemplari possono contenere dati di qualunque tipo. In questo capitolo imparerete a realizzare vostre classi generiche.

16.1 Classi generiche e tipi parametrici

La *programmazione generica* consiste nella creazione di costrutti di programmazione che possano essere utilizzati con molti tipi di dati diversi. Ad esempio, i programmatori della libreria Java che hanno realizzato la classe `ArrayList` hanno sfruttato le tecniche della programmazione generica: come risultato, è possibile creare vettori che contengano elementi di tipi diversi, come `ArrayList<String>`, `ArrayList<BankAccount>` e così via.

In Java, si può raggiungere l'obiettivo della programmazione generica usando l'ereditarietà oppure i tipi parametrici.

La classe `LinkedList` che abbiamo realizzato nel Paragrafo 14.2 è un altro esempio di programmazione generica, dato che in un esemplare di `LinkedList` si possono memorizzare oggetti di qualsiasi tipo. Tale classe raggiunge l'obiettivo della genericità di utilizzo usando l'*ereditarietà*: usa riferimenti di tipo `Object` ed è, quindi, in grado di memorizzare oggetti di qualsiasi tipo. Al contrario, la classe `ArrayList` è una *classe generica*, cioè una classe con un *tipo parametrico*, avente lo scopo di specificare il tipo degli oggetti che vi volete memorizzare. Osservate che solamente la nostra realizzazione di `LinkedList`, vista nel Capitolo 14, usa l'ereditarietà; la libreria standard di Java contiene una classe `LinkedList` che usa tipi parametrici.

Una classe generica ha uno o più tipi parametrici.

Nella dichiarazione di una classe generica, occorre specificare una variabile di tipo per ogni tipo parametrico. Ecco come viene dichiarata la classe `ArrayList` nella libreria standard di Java, usando la *variabile di tipo* `E` per rappresentare il tipo degli elementi:

```
public class ArrayList<E>
{
   public ArrayList() {...}
   public void add(E element) {...}
   ...
}
```

In questo caso, `E` è una variabile di tipo, non una parola riservata di Java; invece di `E` potreste usare un nome diverso, come `ElementType`, ma per le variabili di tipo si è soliti usare nomi brevi e composti di lettere maiuscole.

I tipi parametrici possono essere sostituiti, all'atto della creazione di esemplari, con nomi di classi o di interfacce.

Per poter usare una classe generica, dovete fornire un tipo effettivo che sostituisca il tipo parametrico: si può usare il nome di una classe oppure di un'interfaccia, come in questi esempi:

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

Non si può, però, sostituire un tipo parametrico con uno degli otto tipi di dati primitivi, quindi sarebbe un errore creare un oggetto di tipo `ArrayList<double>`: usate la corrispondente classe involucro, `ArrayList<Double>`.

Quando create un esemplare di una classe generica, il tipo di dato che indicate va a sostituire tutte le occorrenze della variabile di tipo utilizzata nella dichiarazione della classe. Ad 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)
```

Fate un confronto con la dichiarazione del metodo `add` della classe `LinkedList`:

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

Il metodo `add` della classe generica `ArrayList` è più sicuro: è impossibile aggiungere un oggetto di tipo `String` a un esemplare di `ArrayList<BankAccount>`, mentre è possibile aggiungere un oggetto di tipo `String` a 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(); // per oggetti di tipo BankAccount
accounts1.add("my savings"); // errore durante la compilazione
accounts2.add("my savings"); // errore non individuato dal compilatore
```

Il secondo esempio provocherà il lancio di un'eccezione di tipo `ClassCastException` quando qualche altra porzione di codice riceverà una stringa, credendo di ricevere un conto bancario:

```
BankAccount account = (BankAccount) accounts2.getFirst(); // errore
                                                          // in esecuzione
```

I tipi parametrici rendono più sicuro e di più facile comprensione il codice generico.

Il codice che usa la classe generica `ArrayList` è anche di più facile comprensione: quando vedete scritto `ArrayList<BankAccount>`, siete immediatamente consapevoli del fatto che quel vettore deve contenere conti bancari; quando vedete, invece, un esemplare di `LinkedList`, dovete analizzare il codice per scoprire che tipo di dati contenga.

Nei Capitoli 14 e 15, per realizzare strutture dati (liste concatenate, tabelle hash e alberi binari) che fossero dotate di genericità abbiamo usato l'ereditarietà, perché avevate già acquisito familiarità con tale concetto. L'uso di tipi parametrici richiede una nuova sintassi e nuove tecniche di programmazione, che saranno argomento di questo capitolo.

Auto-valutazione

1. La libreria standard mette a disposizione la classe `HashMap<K, V>`, dove `K` è il tipo della chiave e `V` è il tipo del valore. Come esemplare di tale classe, costruite una mappa che memorizzi associazioni tra stringhe e numeri interi.
2. La classe presentata nel Capitolo 15 per realizzare un albero di ricerca binario è un esempio di programmazione generica, perché la potete utilizzare con oggetti che siano esemplari di qualunque classe che realizzi l'interfaccia `Comparable`. Si è, in quel caso, ottenuta la genericità mediante l'ereditarietà o i tipi parametrici?

16.2 Realizzare tipi generici

In questo paragrafo imparerete a realizzare vostre classi generiche. Inizieremo con una classe generica molto semplice, che memorizza *coppie* di oggetti, ciascuno dei quali può essere di tipo qualsiasi. Ad esempio:

```
Pair<String, Integer> result = new Pair<String, Integer>("Harry Morgan", 1729);
```

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

```
String name = result.getFirst();
Integer number = 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 `Integer`, mentre può restituire un singolo oggetto di tipo `Pair<String, Integer>`.

La classe generica `Pair` richiede due tipi parametrici, uno per il tipo del primo elemento e uno per il tipo del secondo elemento.

Dobbiamo scegliere le variabili per questi tipi parametrici. Solitamente per le variabili di tipo si usano nomi brevi e composti di sole lettere maiuscole, come in questi esempi:

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

Le variabili di tipo di una classe generica vanno indicate dopo il nome della classe e sono racchiuse tra parentesi angolari.

Le variabili di tipo di una classe generica vanno indicate dopo il nome della classe, racchiuse tra parentesi angolari:

```
public class Pair<T, S>
```

Per indicare i tipi generici delle variabili di esemplare, dei parametri dei metodi e dei valori da essi restituiti, usate le variabili di tipo.

Nelle dichiarazioni delle variabili di esemplare e dei metodi della classe `Pair`, usiamo la variabile di tipo `T` per indicare il tipo del primo elemento e la variabile di tipo `S` per il tipo del secondo elemento:

```
public class Pair<T, S>
{
   private T first;
   private S second;

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

Alcuni trovano più semplice partire dalla dichiarazione di una classe normale, scegliendo tipi effettivi al posto delle variabili di tipo, come in questo esempio:

```
public class Pair // iniziamo con una coppia di String e Integer
{
```

```
   private String first;
   private Integer second;

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

A questo punto è facile sostituire tutti i tipi `String` con la variabile di tipo `S` e tutti i tipi `Integer` con la variabile di tipo `T`.

Ciò completa la definizione della classe generica `Pair`, che ora è pronta per essere utilizzata ovunque abbiate bisogno di creare una coppia composta da due oggetti di tipo qualsiasi. L'esempio che segue mostra come usare un oggetto di tipo `Pair` per progettare un metodo che restituisca due valori.

File ch16/pair/Pair.java

```
/**
   Questa classe memorizza una coppia di elementi di tipi diversi.
*/
public class Pair<T, S>
{
   private T first;
   private S second;
```

Sintassi di Java 16.1 Dichiarazione di una classe generica

Sintassi

```
modalitàDiAccesso class NomeClasseGenerica<VariabileDiTipo1,
                                           VariabileDiTipo2,...>
{
   variabili di esemplare
   costruttori
   metodi
}
```

Esempio

Specificate una variabile per ogni tipo parametrico.

```
public class Pair<T, S>
{
   private T first;
   private S second;
   ...
   public T getFirst() { return first; }
   ...
}
```

Variabili di esemplare di un tipo di dato parametrico.

Metodo che restituisce un valore di tipo parametrico.

```
   /**
      Costruisce una coppia contenente i due elementi ricevuti.
      @param firstElement il primo elemento
      @param secondElement il secondo elemento
   */
   public Pair(T firstElement, S secondElement)
   {
      first = firstElement;
      second = secondElement;
   }

   /**
      Restituisce il primo elemento di questa coppia.
      @return il primo elemento
   */
   public T getFirst() { return first; }

   /**
      Restituisce il secondo elemento di questa coppia.
      @return il secondo elemento
   */
   public S getSecond() { return second; }

   public String toString() { return "(" + first + ", " + second + ")"; }
}
```

File ch16/pair/PairDemo.java

```
public class PairDemo
{
   public static void main(String[] args)
   {
      String[] names = { "Tom", "Diana", "Harry" };
      Pair<String, Integer> result = firstContaining(names, "a");
      System.out.println(result.getFirst());
      System.out.println("Expected: Diana");
      System.out.println(result.getSecond());
      System.out.println("Expected: 1");
   }

   /**
      Restituisce la prima stringa contenente una stringa assegnata,
      oltre al suo indice nell'array.
      @param strings un array di stringhe
      @param sub una stringa
      @return una coppia (strings[i], i), dove strings[i] è la prima
              stringa in strings contenente sub, oppure una coppia
              (null, -1) se non si trovano corrispondenze
   */
   public static Pair<String, Integer> firstContaining(
         String[] strings, String sub)
   {
      for (int i = 0; i < strings.length; i++)
      {
         if (strings[i].contains(sub))
         {
            return new Pair<String, Integer>(strings[i], i);
         }
      }
```

```
      return new Pair<String, Integer>(null, -1);
   }
}
```

Esecuzione del programma

```
Diana
Expected: Diana
1
Expected: 1
```

Auto-valutazione

3. Come usereste la classe generica `Pair` per costruire una coppia contenente le stringhe `"Hello"` e `"World"`?
4. Che differenza c'è tra un oggetto di tipo `ArrayList<Pair<String, Integer>>` e uno di tipo `Pair<ArrayList<String>, Integer>`?

16.3 Metodi generici

Un metodo generico è un metodo avente un tipo parametrico.

Un metodo generico è un metodo che ha un tipo parametrico e si può anche trovare in una classe che, per se stessa, non è generica. Potete pensare a un tale metodo come a un insieme di metodi che differiscono tra loro soltanto per uno o più tipi di dati. Ad esempio, potremmo voler dichiarare un metodo che possa visualizzare un array di qualsiasi tipo:

```
public class ArrayUtil
{
   /**
      Visualizza tutti gli elementi contenuti in un array.
      @param a l'array da visualizzare
   */
   public <T> static void print(T[] a)
   {
      ...
   }
   ...
}
```

Come detto nel paragrafo precedente, spesso è più facile capire come si realizza un metodo generico partendo da un esempio concreto. Questo metodo visualizza tutti gli elementi presenti in un array di *stringhe*.

```
public class ArrayUtil
{
   public static void print(String[] a)
   {
      for (String e : a)
         System.out.print(e + " ");
      System.out.println();
   }
   ...
}
```

I tipi parametrici di un metodo generico vanno scritti tra i modificatori e il tipo del valore restituito dal metodo.

Per trasformare tale metodo in un metodo generico, sostituite il tipo `String` con un tipo parametrico, diciamo `E`, che stia ad indicare il tipo degli elementi dell'array. Aggiungete un elenco di tipi parametrici, racchiuso tra parentesi angolari, tra i modificatori (in questo caso `public` e `static`) e il tipo del valore restituito (in questo caso `void`):

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

Quando invocate il metodo generico, non dovete specificare i tipi effettivi da usare al posto dei tipi parametrici (e in questo aspetto i metodi generici differiscono dalle classi generiche): invocate semplicemente il metodo con i parametri appropriati e il compilatore metterà in corrispondenza i tipi parametrici con i tipi dei parametri. Ad esempio, considerate questa invocazione di metodo:

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

Quando invocate un metodo generico, non dovete specificare esplicitamente i tipi da usare al posto dei tipi parametrici.

Il tipo del parametro `rectangles` è `Rectangle[]`, mentre il tipo della variabile parametro è `E[]`: il compilatore ne deduce che il tipo effettivo da usare per `E` è `Rectangle`.

Questo particolare metodo generico è un metodo statico inserito in una classe normale (non generica), ma potete definire anche metodi generici che non siano statici. Potete, infine, definire metodi generici all'interno di classi generiche.

Come nel caso delle classi generiche, non potete usare tipi primitivi per sostituire tipi parametrici. Il metodo generico `print` può, quindi, visualizzare array di qualsiasi tipo, *eccetto* array di uno degli otto tipi primitivi. Ad esempio, non si può usare il metodo `print` per visualizzare un array di tipo `int[]`, ma questo non è un grande problema: realizzate semplicemente, oltre al metodo generico `print`, un metodo `print(int[] a)`.

Auto-valutazione

5. Cosa fa esattamente il metodo generico `print` quando fornite come parametro un array di oggetti di tipo `BankAccount` contenente due conti bancari aventi saldo uguale a zero?
6. Il metodo `getFirst` della classe `Pair` è un metodo generico?

16.4 Vincolare i tipi parametrici

I tipi parametrici possono essere soggetti a vincoli.

Spesso è necessario specificare quali tipi possano essere usati in una classe generica oppure in un metodo generico. Considerate, ad esempio, un metodo generico, `min`, che abbia il compito di trovare l'elemento di valore minimo presente in un array di oggetti. Come è possibile trovare l'elemento di valore minimo quando non si ha alcuna informazione in merito al tipo degli elementi? Serve un meccanismo che consenta di confrontare gli elementi dell'array. Una soluzione consiste nel richiedere che gli elementi appartengano a un tipo che realizza l'interfaccia `Comparable`. In una situazione come questa, dobbiamo quindi *vincolare* il tipo parametrico.

Sintassi di Java 16.2 Dichiarazione di un metodo generico

Sintassi

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

Esempio

Specificate la variabile di tipo prima del tipo restituito.

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

Variabile locale di un tipo di dato parametrico.

```
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 al tipo `Comparable` è necessaria per poter invocare il metodo `compareTo`: se non fosse stato specificato il vincolo, il metodo `min` non sarebbe stato compilato, perché non sarebbe stato lecito invocare `compareTo` con l'oggetto `a[i]`, del cui tipo non si avrebbe avuto alcuna informazione (in realtà, la stessa interfaccia `Comparable` è un tipo generico, ma per semplicità l'abbiamo utilizzata senza fornire un tipo come parametro; per maggiori informazioni, consultate Argomenti avanzati 16.1).

Vi capiterà raramente di dover indicare due o più vincoli. In tal caso, separateli con il carattere `&`, come in questo esempio:

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

La parola riservata `extends`, quando viene applicata ai tipi parametrici, significa in realtà "estende o implementa"; i vincoli possono essere classi o interfacce e i tipi parametrici possono essere sostituiti con il tipo effettivo di una classe o di un'interfaccia.

Auto-valutazione

7. Come vincolereste il tipo parametrico di una classe `BinarySearchTree` generica?
8. Modificate il metodo `min` in modo che identifichi l'elemento minimo all'interno di un array di elementi che realizzano l'interfaccia `Measurable` vista nel Capitolo 9.

Errori comuni 16.1

Genericità e ereditarietà

Se `SavingsAccount` è una sottoclasse di `BankAccount`, allora `ArrayList<SavingsAccount>` è una sottoclasse di `ArrayList<BankAccount>`? Anche se forse ne sarete sorpresi, la risposta è no: il legame di ereditarietà presente fra i tipi parametrici non genera un legame di ereditarietà fra le classi generiche corrispondenti e, quindi, non esiste alcuna relazione di ereditarietà tra `ArrayList<SavingsAccount>` e `ArrayList<BankAccount>`.

Questa limitazione è assolutamente necessaria per consentire la verifica della corrispondenza tra i tipi. Immaginate, infatti, che fosse possibile assegnare un oggetto di tipo `ArrayList<SavingsAccount>` a una variabile di tipo `ArrayList<BankAccount>`, in questo modo:

```
ArrayList<SavingsAccount> savingsAccounts
           = new ArrayList<SavingsAccount>();
// quanto segue non è lecito, ma supponiamo che lo sia
ArrayList<BankAccount> bankAccounts = savingsAccounts;
BankAccount harrysChecking = new CheckingAccount();
      // CheckingAccount è una diversa sottoclasse di BankAccount
bankAccounts.add(harrysChecking);
      // va bene, si possono aggiungere oggetti di tipo BankAccount
```

Ma `bankAccounts` e `savingsAccounts` fanno riferimento al medesimo vettore! Se l'assegnazione indicata in grassetto fosse lecita, saremmo in grado di aggiungere un oggetto di tipo `CheckingAccount` a un contenitore di tipo `ArrayList<SavingsAccount>`.

In molte situazioni queste limitazioni possono essere superate usando un carattere jolly (*wildcard*), come descritto in Argomenti avanzati 16.1.

Argomenti avanzati 16.1

Tipi con carattere jolly (*wildcard*)

Spesso si ha la necessità di formulare vincoli un po' complessi per i tipi parametrici: per questo scopo sono stati inventati i tipi con carattere jolly (*wildcard*), che può essere usato in tre diversi modi.

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

Un tipo specificato con carattere jolly è un tipo che può rimanere sconosciuto. Ad esempio, nella classe `LinkedList<E>` si può definire il metodo seguente, che aggiunge alla fine della lista concatenata tutti gli elementi contenuti in `other`.

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

```
      while (iter.hasNext()) add(iter.next());
   }
```

Il metodo `addAll` non richiede che il tipo degli elementi di `other` sia un qualche tipo specifico: consente l'utilizzo di qualsiasi tipo che sia un sottotipo di `E`. Ad esempio, potete usare `addAll` per aggiungere a un esemplare di `LinkedList<BankAccount>` tutti gli elementi contenuti in un esemplare di `LinkedList<SavingsAccount>`.

Per vedere un tipo *wildcard* con vincolo di tipo `super`, torniamo al metodo `min` del paragrafo precedente. Ricordate che `Comparable` è un'interfaccia generica e il tipo che la rende generica serve a specificare il tipo del parametro del metodo `compareTo`.

```
public interface Comparable<T>
{
   int compareTo(T other);
}
```

Di conseguenza, potremmo voler specificare un tipo vincolato:

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

Questo vincolo, però, è troppo restrittivo. Immaginate che la classe `BankAccount` realizzi l'interfaccia `Comparable<BankAccount>`: di conseguenza, anche la sua sottoclasse `SavingsAccount` realizza `Comparable<BankAccount>` e *non* `Comparable<SavingsAccount>`. Se volete usare il metodo `min` con un array di tipo `SavingsAccount[]`, allora il tipo che specifica il parametro dell'interfaccia `Comparable` deve essere *qualsiasi supertipo* del tipo di elemento dell'array:

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

Ecco, invece, un esempio di carattere jolly che specifica l'assenza di vincoli. La classe `Collections` definisce il metodo seguente:

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

Una dichiarazione di questo tipo è, in pratica, un'abbreviazione per la seguente:

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

16.5 Cancellazione dei tipi (*type erasure*)

La macchina virtuale elimina i tipi parametrici, sostituendoli con i loro vincoli o con `Object`.

Dato che i tipi generici sono stati introdotti nel linguaggio Java soltanto di recente, la macchina virtuale che esegue programmi Java non lavora con classi o metodi generici: i tipi parametrici vengono "cancellati", cioè sostituiti da tipi Java ordinari. Ciascun tipo parametrico è sostituito dal relativo vincolo, oppure da `Object` se non è vincolato.

Ad esempio, la classe generica `Pair<T, S>` viene sostituita dalla seguente classe "grezza" (*raw*):

```
public class Pair
{
   private Object first;
   private Object second;
```

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

Come potete vedere, i tipi parametrici, `T` e `S`, sono stati sostituiti da `Object`; il risultato è una classe ordinaria.

Ai metodi generici viene applicato il medesimo procedimento. Dopo la cancellazione dei tipi parametrici, il metodo `min` del paragrafo precedente si trasforma in un metodo ordinario; notate come, in questo esempio, il tipo parametrico sia sostituito dal suo vincolo, l'interfaccia `Comparable`.

```
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;
}
```

Non si possono costruire oggetti o array di un tipo generico.

Conoscere l'esistenza dei tipi grezzi aiuta a capire i limiti della programmazione generica in Java. Ad esempio, non potete costruire esemplari di un tipo generico. Il seguente metodo, che tenta di riempire un array con copie di oggetti predefiniti, sarebbe sbagliato:

```
public static <E> void fillWithDefaults(E[] a)
{
   for (int i = 0; i < a.length; i++)
      a[i] = new E(); // ERRORE
}
```

Per capire per quale motivo ciò costituisca un problema, eseguite il procedimento di cancellazione dei tipi parametrici, come se foste il compilatore:

```
public static void fillWithDefaults(Object[] a)
{
   for (int i = 0; i < a.length; i++)
      a[i] = new Object(); // inutile
}
```

Ovviamente, se costruite un array di tipo `Rectangle[]`, non volete che il metodo lo riempia di esemplari di `Object`, ma, dopo la cancellazione dei tipi parametrici, questo è ciò che farebbe il codice che abbiamo scritto.

In situazioni come questa, il compilatore segnala un errore, per cui dovete necessariamente trovare un modo diverso per risolvere il vostro problema. In questo particolare esempio, potete fornire uno specifico oggetto predefinito:

```
public static <E> void fillWithDefaults(E[] a, E defaultValue)
{
   for (int i = 0; i < a.length; i++)
```

```
      a[i] = defaultValue;
}
```

Analogamente, non è possibile costruire un array di un tipo generico.

```
public class Stack<E>
{
   private E[] elements;
   ...
   public Stack()
   {
      elements = new E[MAX_SIZE]; // ERRORE
   }
}
```

Dato che l'espressione che costruisce l'array, `new E[]`, diventerebbe, dopo la cancellazione dei tipi parametrici, `new Object[]`, il compilatore non la consente. Come soluzione, si può usare un vettore:

```
public class Stack<E>
{
   private ArrayList<E> elements;
   ...
   public Stack()
   {
      elements = new ArrayList<E>(); // così va bene
   }
}
```

oppure si può usare un array di `Object`, inserendo un cast ogni volta che si legge un valore contenuto nell'array:

```
public class Stack<E>
{
   private Object[] elements;
   private int size;
   ...
   public Stack()
   {
      elements = new Object[MAX_SIZE]; // anche così va bene
   }
   ...
   public E pop()
   {
      size--;
      return (E) elements[size];
   }
}
```

Il cast genera un *avvertimento* (*warning*) in fase di compilazione perché non è verificabile.

Queste limitazioni sono, francamente, imbarazzanti: ci auguriamo che le future versioni di Java non effettuino più la cancellazione dei tipi parametrici, in modo da poter eliminare le attuali restrizioni che ne sono, appunto, conseguenza.

Auto-valutazione

9. Cosa si ottiene applicando la cancellazione dei tipi parametrici al metodo `print` visto nel Paragrafo 16.3?
10. Si potrebbe realizzare una pila in questo modo?

```
public class Stack<E>
{
   private E[] elements;
   ...
   public Stack()
   {
      elements = (E[]) new Object[MAX_SIZE];
   }
}
```

Errori comuni 16.2

Usare tipi generici in un contesto statico

Non si possono usare tipi parametrici per dichiarare variabili statiche, metodi statici o classi interne statiche. Ad esempio, quanto segue non è lecito:

```
public class LinkedList<E>
{
   private static E defaultValue; // ERRORE
   ...
   public static List<E> replicate(E value, int n) {...} // ERRORE
   private static class Node { public E data; public Node next; } // ERRORE
}
```

Nel caso di variabili statiche, questa limitazione è molto stringente. Dopo la cancellazione dei tipi generici, esiste un'unica variabile, `LinkedList.defaultValue`, mentre la dichiarazione della variabile statica lascerebbe falsamente intendere che ne esista una diversa per ogni diverso tipo di `LinkedList<E>`.

Per i metodi statici e le classi interne statiche esiste una semplice alternativa: aggiungere un tipo parametrico.

```
public class LinkedList<E>
{
   ...
   public static <T> List<T> replicate(T value, int n) {...} // va bene
   private static class Node<T> { public T data; public Node<T> next; }
         // va bene
}
```

Riepilogo degli obiettivi di apprendimento

Classi generiche e tipi parametrici

- In Java, si può raggiungere l'obiettivo della programmazione generica usando l'ereditarietà oppure i tipi parametrici.

- Una classe generica ha uno o più tipi parametrici.
- I tipi parametrici possono essere sostituiti, all'atto della creazione di esemplari, con nomi di classi o di interfacce.
- I tipi parametrici rendono più sicuro e di più facile comprensione il codice generico.

Realizzazione di classi e interfacce generiche

- Le variabili di tipo di una classe generica vanno indicate dopo il nome della classe e sono racchiuse tra parentesi angolari.
- Per indicare i tipi generici delle variabili di esemplare, dei parametri dei metodi e dei valori da essi restituiti, usate le variabili di tipo.

Realizzazione di metodi generici

- Un metodo generico è un metodo avente un tipo parametrico.
- I tipi parametrici di un metodo generico vanno scritti tra i modificatori e il tipo del valore restituito dal metodo.
- Quando invocate un metodo generico, non dovete specificare esplicitamente i tipi da usare al posto dei tipi parametrici.

Espressione di vincoli per i tipi parametrici

- I tipi parametrici possono essere soggetti a vincoli.

Limitazioni sulla programmazione generica in Java imposte dalla cancellazione dei tipi parametrici

- La macchina virtuale elimina i tipi parametrici, sostituendoli con i loro vincoli o con `Object`.
- Non si possono costruire oggetti o array di un tipo generico.

Esercizi di ripasso

* **Esercizio R16.1.** Cos'è un tipo parametrico?

* **Esercizio R16.2.** Qual è la differenza tra una classe generica e una classe ordinaria?

* **Esercizio R16.3.** Qual è la differenza tra una classe generica e un metodo generico?

* **Esercizio R16.4.** Nella libreria standard di Java, identificate un esempio di metodo generico non statico.

** **Esercizio R16.5.** Nella libreria standard di Java, identificate quattro esempi di classi generiche che abbiano due tipi parametrici.

** **Esercizio R16.6.** Nella libreria standard di Java, identificate un esempio di classe generica che non sia una delle classi che realizzano contenitori.

* **Esercizio R16.7.** Perché nel metodo seguente serve un vincolo per il tipo parametrico, `T`?

```
<T extends Comparable> int binarySearch(T[] a, T key)
```

** **Esercizio R16.8.** Perché nella classe `HashSet<E>` non serve un vincolo per il tipo parametrico, `E`?

* **Esercizio R16.9.** Che cosa rappresenta un esemplare di `ArrayList<Pair<T, T>>`?

** **Esercizio R16.10.** Illustrate i vincoli applicati ai tipi nel seguente metodo della classe `collections`:

```
public static <T extends Comparable<? super T>> void sort(List<T> a)
```

Perché non è sufficiente scrivere `<T extends Comparable>` oppure `<T extends Comparable<T>>`?

★ **Esercizio R16.11.** Cosa succede quando si fornisce un oggetto di tipo `ArrayList<String>` a un metodo che riceve un parametro di tipo `ArrayList`? Provate e fornite una spiegazione.

★★★ **Esercizio R16.12.** Cosa succede quando si fornisce un oggetto di tipo `ArrayList<String>` a un metodo che riceve un parametro di tipo `ArrayList` e memorizza in tale vettore un oggetto di tipo `BankAccount`? Provate e fornite una spiegazione.

★★ **Esercizio R16.13.** Che risultato si ottiene con la seguente verifica di condizione?

```
ArrayList<BankAccount> accounts = new ArrayList<BankAccount>();
if (accounts instanceof ArrayList<String>) ...
```

Provate e fornite una spiegazione.

★★★ **Esercizio R16.14.** La classe `ArrayList<E>`della libreria standard di Java deve gestire un array di oggetti di tipo `E`, ma, in Java, non è lecito costruire un array generico, di tipo `E[]`. Osservate, nel codice sorgente della libreria, che fa parte del JDK, la soluzione che è stata adottata e fornite una spiegazione.

Risposte alle domande di auto-valutazione

1. `HashMap<String, Integer>`
2. Mediante l'ereditarietà.
3. `new Pair<String, String>("Hello", "World")`
4. Il primo contiene coppie, ad esempio `[(Tom, 1), (Harry, 3)]`, mentre il secondo contiene un elenco di stringhe e un unico numero intero, ad esempio `([Tom, Harry], 1)`.
5. Ciò che viene visualizzato dipende dalla definizione del metodo `toString` nella classe `BankAccount`.
6. No, il metodo non ha tipi parametrici, si tratta di un metodo ordinario all'interno di una classe generica.
7.
```
public class BinarySearchTree
      <E extends Comparable>
```
8.
```
public static <E extends Measurable>
      E min(E[] a)
{
   E smallest = a[0];
   for (int i = 1; i < a.length; i++)
      if (a[i].getMeasure() <
            smallest.getMeasure())
         smallest = a[i];
   return smallest;
}
```
9.
```
public static void print(Object[] a)
{
   for (Object e : a)
      System.out.print(e + " ");
   System.out.println();
}
```

10. La classe supera la compilazione (con un *warning*), ma si tratta di una tecnica debole e se, in futuro, non si effettuerà più la cancellazione dei tipi parametrici, questo codice sarà *errato*: il cast da `Object[]` a `String[]` provocherà il lancio di un'eccezione.

Sul sito web dedicato al libro si trova una vasta raccolta di esercizi di programmazione e di progetti più complessi.