15

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.

15.1 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.

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)
```

15.2 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<>("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 Pair.java

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

   /**
      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 + ")"; }
}
```

Sintassi di Java 15.1 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.

File 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<>(strings[i], i);
         }
```

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

Esecuzione del programma

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

15.3 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 static <T> 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 rappresenti 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)`.

15.4 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 vettore 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 del vettore. Nel Capitolo 10 abbiamo progettato un'interfaccia che aveva proprio tale obiettivo:

```
public interface Measurable
{
   double getMeasure();
}
```

Possiamo vincolare il tipo ammissibile per gli elementi del vettore, richiedendo che sia un tipo che implementa l'interfaccia `Measurable`. In Java, questo si ottiene aggiungendo la clausola `extends Measurable` dopo il tipo parametrico:

```
public static <E extends Measurable> double average(ArrayList<E> objects)
```

Sintassi di Java 15.2 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.

Questo significa "il tipo `E` oppure una delle sue superclassi estende o implementa `Measurable`", per cui diciamo che `E` è un sottotipo di `Measurable`.

Ecco il codice completo del metodo `average`:

```
public static <E extends Measurable> double average(ArrayList<E> objects)
{
   if (objects.size() == 0) { return 0; }
   double sum = 0;
   for (E obj : objects)
   {
      sum = sum + obj.getMeasure();
   }
   return sum / objects.size();
}
```

Osserviamo l'invocazione `obj.getMeasure()`: la variabile `obj` è di tipo `E`, che è un sottotipo di `Measurable`, quindi siamo certi che sia possibile invocare il metodo `getMeasure` con `obj`.

Se la classe `BankAccount` implementasse l'interfaccia `Measurable`, allora potremmo invocare il metodo `average` con un vettore di oggetti di tipo `BankAccount`, mentre non possiamo farlo con un vettore di stringhe perché la classe `String` non implementa l'interfaccia `Measurable`.

Consideriamo ora l'obiettivo di trovare il valore minimo in un vettore. Possiamo farlo restituendo l'elemento avente la misura minima, ma l'interfaccia `Measurable` è stata progettata per questo libro e non fa parte della libreria standard. Usiamo, invece, l'interfaccia `Comparable`, implementata da molte classi: si tratta, a sua volta, di un'interfaccia generica, il cui tipo parametrico specifica il tipo della variabile parametro del metodo `compareTo`.

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

Ad esempio, la classe `String` implementa `Comparable<String>`: si possono confrontare stringhe con altre stringhe, ma non con esemplari di classi diverse.

Se il vettore contiene elementi di tipo `E`, vogliamo porre il vincolo che il tipo `E` implementi `Comparable<E>`. Ecco il metodo:

```
public static <E extends Comparable<E>> E min(ArrayList<E> objects)
{
   E smallest = objects.get(0);
   for (int i = 1; i < objects.size(); i++)
   {
      E obj = objects.get(i);
      if (obj.compareTo(smallest) < 0)
      {
         smallest = obj;
      }
   }
   return smallest;
}
```

Grazie al tipo generico vincolato, sappiamo che l'oggetto `obj` ha il metodo

```
int compareTo(E other)
```

Quindi, l'invocazione

```
obj.compareTo(smallest)
```

è valida.

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

```
<E extends Comparable<E> & Measurable>
```

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.

Errori comuni 15.1

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<>();
// 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 15.1.

Errori comuni 15.2

L'eccezione `ArrayStoreException`

Nella sezione Errori comuni 15.1 avete visto come non sia possibile assegnare un vettore di elementi di tipo sottoclasse a un vettore di elementi di tipo superclasse. Ad esempio, non si può usare un oggetto di tipo `ArrayList<SavingsAccount>` al posto di un oggetto di tipo `ArrayList<BankAccount>`.

Si tratta di un divieto che lascia un po' perplessi, perché l'assegnazione equivalente tra array è ammessa. Ad esempio:

```
SavingsAccount[] savingsAccounts = new SavingsAccount[10];
BankAccount[] bankAccounts = savingsAccounts; // OK
```

Abbiamo visto che c'è un valido motivo per cui questo viene impedito nel caso dei vettori: se fosse lecito, riusciremmo a memorizzare un oggetto di tipo `CheckingAccount` in un vettore di `SavingsAccount`.

Proviamo a fare la stessa cosa con gli array:

```
BankAccount harrysChecking = new CheckingAccount();
bankAccounts[0] = harrysChecking; // Lancia ArrayStoreException
```

Questo codice viene compilato, perché l'oggetto a cui fa riferimento `harrysChecking` è di tipo `CheckingAccount` e, quindi, può essere memorizzato in un array di `BankAccount`. Ma `bankAccounts` e `savingsAccounts` fanno, in realtà, riferimento al medesimo array, che è di tipo `SavingsAccount[]`: quando il programma viene eseguito, l'array si rifiuta di accogliere un oggetto di tipo `CheckingAccount` e l'interprete lancia `ArrayStoreException`.

Quindi, tanto i vettori quanto gli array evitano di commettere errori con i tipi, ma lo fanno in modi diversi. La classe `ArrayList` li evita durante la compilazione, mentre gli array lo fanno al momento dell'esecuzione. In generale si preferisce la segnalazione di errori al momento della compilazione, ma la strada è veramente in salita, come potrete vedere nella sezione Argomenti avanzati 15.1: bisogna faticare molto per spiegare con precisione al compilatore quali siano le conversioni consentite.

Argomenti avanzati 15.1

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 superiore	`? extends B`	Qualsiasi sottotipo di B
Vincolo con limite inferiore	`? 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`, esaminiamo nuovamente il metodo `min`:

```
public static <E extends Comparable<E>> E min(ArrayList<E> objects)
```

Il vincolo così espresso è, in realtà, troppo restrittivo. Supponiamo che la classe `BankAccount` implementi l'interfaccia `Comparable<BankAccount>`. In tal caso, la sua sottoclasse `SavingsAccount` implementerà, a sua volta, la stessa interfaccia, `Comparable<BankAccount>`, e *non* `Comparable<SavingsAccount>`. Se vogliamo poter utilizzare il metodo `min` con un vettore di `SavingsAccount`, il tipo parametrico dell'interfaccia `Comparable` deve essere *qualsiasi supertipo* del tipo specificato per gli elementi del vettore:

```
public static <E extends Comparable<? super E>> E min(ArrayList<E> objects)
```

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)
```

La sezione Errori comuni 15.2 confronta questa limitazione con il comportamento degli array in Java, apparentemente più permissivo.

15.5 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. Consideriamo il metodo seguente:

```
public static <E extends Measurable> E min(E[] objects)
{
   E smallest = objects[0];
   for (int i = 1; i < objects.length; i++)
   {
      E obj = objects[i];
      if (obj.getMeasure() < smallest.getMeasure())
      {
         smallest = obj;
      }
   }
   return smallest;
}
```

La cancellazione del tipo parametrico prevede la sua sostituzione con il suo vincolo, l'interfaccia `Measurable`:

```
public static Measurable min(Measurable[] objects)
{
   Measurable smallest = objects[0];
```

```
      for (int i = 1; i < objects.length; i++)
      {
         Measurable obj = objects[i];
         if (obj.getMeasure() < smallest.getMeasure())
         {
            smallest = obj;
         }
      }
      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<>(); // 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.

Errori comuni 15.3

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
   static class Node { public E data; public Node next; } // ERRORE
}
```

Nel caso di variabili statiche, questa limitazione è molto ragionevole. 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
   static class Node<T> { public T data; public Node<T> next; } // va bene
}
```

Riepilogo degli obiettivi di apprendimento

Classi generiche e 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.

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 riepilogo e approfondimento

* **R15.1.** Cos'è un tipo parametrico?

* **R15.2.** Qual è la differenza tra una classe generica e una classe ordinaria?

* **R15.3.** Qual è la differenza tra una classe generica e un metodo generico?

** **R15.4.** Perché è necessario fornire argomenti di tipo quando si crea un esemplare di una classe generica, mentre non lo si deve fare quando si utilizza un metodo generico?

* **R15.5.** Nella libreria standard di Java, identificate un esempio di metodo generico non statico.

** **R15.6.** Nella libreria standard di Java, identificate quattro esempi di classi generiche che abbiano due tipi parametrici.

** **R15.7.** Nella libreria standard di Java, identificate un esempio di classe generica che non sia una delle classi che realizzano contenitori.

* **R15.8.** Perché nel metodo seguente serve un vincolo per il tipo parametrico, `T`?

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

** **R15.9.** Perché nella classe `HashSet<E>` non serve un vincolo per il tipo parametrico, `E`?

* **R15.10.** Che cosa rappresenta un esemplare di `ArrayList<Pair<T, T>>`?

** **R15.11.** 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>>`?

* **R15.12.** Cosa succede se si passa un oggetto di tipo `ArrayList<String>` a un metodo che ha una variabile parametro di tipo `ArrayList`? Provate e fornite una spiegazione.

*** **R15.13.** Cosa succede se si passa un oggetto di tipo `ArrayList<String>` a un metodo che ha una variabile parametro di tipo `ArrayList` e che, nel vettore ricevuto, inserisce un oggetto di tipo `BankAccount`? Provate e fornite una spiegazione.

** **R15.14.** Che risultato si ottiene con la seguente verifica di condizione?

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

Provate e fornite una spiegazione.

** **R15.15.** 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.

Esercizi di programmazione

* **E15.1.** Modificate la classe generica `Pair` in modo che i due valori siano dello stesso tipo.

* **E15.2.** Aggiungete alla classe `Pair` dell'esercizio precedente un metodo `swap` che scambi tra loro il primo e il secondo elemento della coppia.

** **E15.3.** Realizzate un metodo statico generico `PairUtil.swap`, il cui parametro sia un oggetto di tipo `Pair`, usando la classe generica definita nel Paragrafo 15.2. Il metodo deve restituire una nuova coppia avente il primo ed il secondo elemento scambiati rispetto alla coppia originaria.

* **E15.4.** Realizzate un metodo generico che, dato un oggetto di tipo `Map<K, V>`, restituisce un oggetto di tipo `List<Pair<K, V>>` contenente le coppie chiave/valore presenti nella mappa.

** **E15.5.** Realizzate una versione generica dell'algoritmo di ricerca binaria.

** **E15.6.** Realizzate una versione generica dell'algoritmo di ordinamento per selezione.

*** **E15.7.** Realizzate una versione generica dell'algoritmo di ordinamento per fusione. Il programma deve essere compilato senza che vengano generati *warning*.

E15.8. [*omesso*]

E15.9. [*omesso*]

** **E15.10.** Dotate la classe `Pair` vista nel Paragrafo 15.2 dei metodi `equals` e `hashCode` appropriati e realizzate una classe `HashMap` che usi un esemplare di `HashSet<Pair<K, V>>`.

*** **E15.11.** Realizzate una versione generica del generatore di permutazioni visto nel Paragrafo 12.4, in modo che generi tutte le permutazioni degli elementi presenti in un oggetto di tipo `List<E>`.

** **E15.12.** Scrivete un metodo statico generico, `print`, che visualizzi gli elementi di qualsiasi oggetto che implementi l'interfaccia `Iterable<E>`, separandoli con virgole. Inserite tale metodo in una classe appropriata.

E15.13. [*omesso*]

** **E15.14.** Trasformate l'interfaccia `Measurer` vista nel Paragrafo 10.4 in un'interfaccia generica. Realizzate, poi, il metodo statico `T max(T[] values, Measurer<T> meas)`.

* **E15.15.** Realizzate il metodo statico `void append(ArrayList<T> a, ArrayList<T> b)` che aggiunga gli elementi di `b` in fondo al vettore `a`.

** **E15.16.** Modificate il metodo realizzato nell'esercizio precedente in modo che il secondo vettore possa contenere elementi di qualsiasi sottoclasse del tipo definito per gli elementi del primo vettore. Ad esempio, se `people` è un oggetto di tipo `ArrayList<Person>` e `students` è un oggetto di tipo `ArrayList<Student>`, con `Student` sottoclasse di `Person`, allora `append(people, students)` deve essere compilato correttamente, ma `append(students, people)` no.

** **E15.17.** Modificate il metodo realizzato nell'Esercizio E15.15 in modo che lasci inalterato il primo vettore e, invece, restituisca un nuovo vettore contenente gli elementi di entrambi i vettori ricevuti come parametri.

** **E15.18.** Modificate il metodo realizzato nell'esercizio precedente in modo che riceva e restituisca array, non vettori. *Suggerimento*: `Arrays.copyOf`.

* **E15.19.** Realizzate un metodo statico che inverta gli elementi all'interno di un vettore generico.

* **E15.20.** Realizzate un metodo statico che restituisca un vettore contenente in ordine inverso gli elementi presenti nel vettore generico ricevuto come parametro, senza modificarlo.

** **E15.21.** Realizzate un metodo statico che verifichi se un vettore generico è palindromo, cioè se i valori presenti nelle posizioni di indice `i` e `n - 1 - i` sono uguali, per qualunque valore di `i`, essendo `n` la dimensione del vettore.

** **E15.22.** Realizzate un metodo statico che verifichi se gli elementi di un vettore generico sono disposti al suo interno in ordine crescente. Gli elementi devono essere confrontabili.