Dal linguaggio C al linguaggio Java

(Quarta parte)

Riccardo Silvestri

11-5-2009 17-5-2010

Sommario della quarta parte

Genericità

La genericità in Java può essere sinteticamente descritta come uno strumento per trattare i tipi in modo parametrico. L'aggettivo *generico* è proprio usato per denotare un tipo o un metodo la cui definizione dipende da una o più variabili di tipo. In questo modo la definizione del tipo o metodo è generica perchè può essere adattata semplicemente sostituendo le variabili di tipo con tipi specifici. L'istanziamento delle variabili di tipo, come vedremo, può essere esplicitamente specificata dal programmatore o è automaticamente inferita dal compilatore. Tutto ciò ha un grande vantaggio. Permette di trattare in modo uniforme e sintetico tipi e metodi che altrimenti sarebbero dovuti essere specificatamente definiti in tantissime versioni tutte molto simili tra loro. E questo a sua volta migliora la leggibilità, la riusabilità e facilita la manutenzione del codice.

Inizieremo descrivendo le conversioni boxing/unboxing che sono una agevolazione fornita dal compilatore per sfruttare al meglio i benefici della genericità anche per i tipi primitivi. Poi passeremo a discutere i metodi generici e infine i tipi generici.

Boxing e unboxing

In Java, come sappiamo, c'è una netta distinzione tra i tipi primitivi e i tipi riferimento. Il tipo `Object` è un supertipo di tutti i tipi riferimento, siano essi classi, array o interfacce, mentre i tipi primitivi non hanno supertipi. Il tipo `Object` permette di trattare oggetti di tipo differente in modo uniforme. Ad

esempio è possibile definire una classe per insiemi di `Object` che può essere usata per insiemi di oggetti di un qualsiasi tipo riferimento. Ciò non è possibile per i tipi primitivi perché né `Object` né nessun'altro tipo è un supertipo di tutti i tipi primitivi. Inoltre, come vedremo presto, anche la genericità non è direttamente applicabile ai tipi primitivi.

Chiaramente, c'è un modo per sfruttare, almeno indirettamente, l'uniformità offerta dalla classe `Object` e dalla genericità anche per i tipi primitivi: basterà introdurre per ogni tipo primitivo un corrispondente tipo riferimento e usare quest'ultimo al posto del tipo primitivo. Proprio per agevolare questa possibilità, nel package `java.lang`, ci sono otto tipi riferimento che corrispondono agli otto tipi primitivi in accordo alla seguente tabella:

Tipo primitivo	Tipo riferimento
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Inoltre, cosa ancora più importante, il compilatore Java esegue conversioni automatiche dai tipi primitivi a questi tipi riferimento e viceversa, ovunque ciò risulti appropriato. La conversione di un tipo primitivo al corrispondente tipo riferimento è chiamata *boxing*, mentre quella inversa, dal tipo riferimento al corrispondente tipo primitivo, è chiamata *unboxing*. Ecco alcuni esempi:

```
Integer intO;
intO = 13;               // conversione boxing: è equivalente a
intO = new Integer(13);  // questo

int k;
k = intO;                // conversione unboxing: è equivalente a
k = intO.intValue();     // questo

k = intO*intO;              // unboxing
intO = k*intO*(2 + intO);   // unboxing e boxing
Character charO = 'A';      // boxing
char c = charO;             // unboxing
```

Questi esempi riguardano `int` e `char` ma conversioni analoghe sono effettuate per tutti i tipi primitivi. Per chiarire le conversioni relativamente agli array consideriamo i seguenti metodi:

```
public static int sum(Integer[] a) {
    int s = 0;
    for (int i = 0 ; i < a.length ; i++)
        s += a[i];        // unboxing
    return s;
}

public static Integer sumInteger(Integer[] a) {
    return sum(a);        // boxing
}
```

E poi consideriamo i seguenti frammenti di codice:

```
Integer[] integerA = new Integer[] {2, 10003, 13};   // boxing
Integer intO = 13;
int[] intA = new int[] { intO, 2*intO };             // unboxing

intA = integerA;   // ERRORE in compilazione: tipi incompatibili (Integer[] e int[])
integerA = intA;   // ERRORE in compilazione: tipi incompatibili (Integer[] e int[])

int k = sum(integerA);
k = sumInteger(integerA);
k = sum(intA);    // ERRORE in compilazione: tipi incompatibili (Integer[] e int[])
```

Come si vede, le conversioni non si estendono agli array di tipi primitivi o dei loro corrispondenti tipi

riferimento. Inoltre, il seguente esempio mostra che in relazione all'operatore di uguaglianza può accadere qualcosa di inaspettato:

```
if (sum(integerA) == sum(integerA))                  // VERO
    ...
if (sumInteger(integerA) == sumInteger(integerA))    // FALSO
    ...
```

Il secondo test di uguaglianza risulta falso nonostante i valori interi ritornati dalle due invocazioni del metodo `sumInteger()` siano uguali. La ragione è che qui non è stata effettuata la conversione (unboxing) e quindi l'operatore di uguaglianza è stato applicato agli oggetti di tipo `Integer` ritornati dalle due invocazioni. Generalmente questi oggetti sono diversi anche se hanno lo stesso valore intero. In realtà la situazione è più complessa perchè in alcuni casi il test risulterà vero e in altri risulterà falso. Dipende se per rappresentare un certo valore tramite uno degli otto tipi riferimento il compilatore ha usato un singolo oggetto (*caching*) oppure ha usato oggetti diversi. Il caching è garantito per tutti i valori di tipo `byte` e `boolean`, per i valori di tipo `short` e `int` compresi tra -128 e 127 e per i `char` tra `'\u0000'` e `'\u007F'`. Ma il caching è possibile che sia usato anche per altri valori. Data la difficoltà di prevedere se il caching sarà usato o meno, l'uso dell'operatore di uguaglianza per testare l'uguaglianza di valore di oggetti appartenenti ai tipi di riferimento corrispondenti ai tipi primitivi è fortemente sconsigliato.

Gli altri operatori relazionali possono invece essere usati con tranquillità perché questi inducono sempre la conversione unboxing:

```
Integer intO = 12345;
Integer intO2 = 12345;
if (12345 == intO)                    // VERO (unboxing)
    ...
if (intO2 == intO)                    // FALSO (oggetti differenti)
    ...
if ((int)intO2 == intO)               // VERO (unboxing)
    ...
if (intO2 <= intO)                    // VERO (unboxing)
    ...
```

L'esempio mostra che l'operatore di uguaglianza può anch'esso essere usato con tranquillità se si ha l'accortezza di effettuare il cast di almeno uno dei due operandi.

Il package `java.lang` contiene anche una classe astratta `Number` che è un supertipo dei tipi riferimento che corrispondono ai tipi primitivi numerici (`Byte`, `Short`, `Integer`, `Long`, `Float` e `Double`). Ecco alcuni esempi:

```
Number num = 13;          // boxing in Integer
num = intO;
num = 3.14;               // boxing in Double
int k = num;              // ERRORE: tipi incompatibili (int e Number)
double d = num;           // ERRORE: tipi incompatibili (double e Number)
Double dO = num;          // ERRORE: tipi incompatibili (Double e Number)
num = new Number();       // ERRORE: Number è una classe astratta
Number[] numA;
numA = integerA;
numA = intA;              // ERRORE: tipi incompatibili (Number[] e int[])
```

Le conversioni boxing e unboxing sono effettuate anche relativamente ad un qualsiasi supertipo degli otto tipi riferimento della tabella. Così, oltre ad essere applicate in relazione a `Number`, sono applicate anche in relazione ad `Object`:

```
Object obj = 13;          // boxing in Integer
obj = 'a';                // boxing in Character
obj = 3.14;               // boxing in Double
obj = true;               // boxing in Boolean
boolean b;
b = obj;                  // ERRORE: tipi incompatibili (boolean e Object)
b = (boolean)obj;         // ERRORE: tipi non convertibili (boolean e Object)
b = (Boolean)obj;         // OK
```

Si osservi che il primo cast è legale solamente se applicato ad un oggetto di tipo `Boolean` e il secondo cast provoca una conversione unboxing.

Metodi generici

Un metodo generico è un metodo in cui uno o più nomi di tipo sono sostituiti da dei *parametri formali di tipo* (*formal type parameters*) detti anche *variabili di tipo* (*type variables*). Quando il metodo è invocato le variabili di tipo sono sostituite con nomi di tipi specifici, secondo certe regole che vedremo fra poco. Le variabili di tipo devono essere dichiarate tra parentesi angolari, `<...>`, nell'intestazione del metodo subito dopo gli eventuali modificatori e prima del tipo ritornato dal metodo.

Consideriamo un metodo `find()` che preso in input un array e un valore (dello stesso tipo delle componenti dell'array) ritorna l'indice della prima posizione dell'array che contiene il valore e se il valore non è presente ritorna `-1`. Volendo definire il metodo così che possa essere usato per array di un qualsiasi tipo riferimento, lo definiamo generico rispetto al tipo delle componenti dell'array. Definiamo anche, a mo' di confronto, una versione `find2()` che cerca di emulare la genericità usando il tipo `Object`. Inoltre definiamo anche un `main()` per mettere alla prova i due metodi.

```
public class Test {
        // metodo generico: T è la variabile di tipo
    public static <T> int find(T[] a, T x) {
        for (int i = 0 ; i < a.length ; i++)
            if (a[i].equals(x)) return i;
        return -1;
    }

    public static int find2(Object[] a, Object x) {
        for (int i = 0 ; i < a.length ; i++)
            if (a[i].equals(x)) return i;
        return -1;
    }

    public static void main(String[] args) {
        String[] sA = new String[] {"A", "B", "C"};
        Integer[] intA = new Integer[] {12, 23, 1, 234};
        int k;
        k = find(sA, "C");              // T è sostituita con String
        k = find(intA, 2);              // T è sostituita con Integer
        k = find(intA, "A");
            // le stesse invocazioni sono possibili anche con il metodo find2()
        k = find2(sA, "C");
        k = find2(intA, 2);
        k = find2(intA, "A");
    }
}
```

La variabile di tipo `T` è usata come se fosse il nome di un tipo effettivo. La differenza è che non ci sono tipi effettivi che si chiamano `T` e `T` è infatti dichiarata variabile di tipo tramite l'espressione `<T>`. In realtà non è sempre vero che una variabile di tipo può essere usata come se fosse il nome di un tipo effettivo, ci sono infatti alcune importanti eccezioni che vedremo fra poco.

Quando il metodo generico `find()` è invocato il compilatore inferisce il tipo che deve essere "assegnato" alla variabile `T`. Si ricordi che una variabile di tipo può stare solamente per tipi riferimento, non può mai essere sostituita con tipi primitivi (anche per questo sono state introdotte le conversioni boxing/unboxing). Nella prima invocazione, siccome sia il tipo delle componenti dell'array del primo argomento che il tipo del secondo argomento è `String`, il tipo inferito è `String`. Nella seconda invocazione avviene una conversione boxing che converte il secondo argomento in un oggetto di tipo `Integer` che è anche il tipo delle componenti dell'array passato come primo argomento. Ne deriva che il tipo inferito è `Integer`. Nella terza invocazione i tipi relativi ai due argomenti sono differenti, il primo è `Integer` e l'altro è `String`. In questi casi il tipo inferito dal compilatore è il supertipo comune più "vicino" ai tipi relativi agli argomenti (in questo caso il tipo inferito è `Serializable & Comparable<?>`).

Le stesse invocazioni sono anche lecite per `find2()`. Allora, qual'è il vantaggio dell'uso della genericità? La versione generica del metodo permette di stabilire il tipo da "assegnare" alla variabile `T` al momento dell'invocazione. Il tipo da assegnare deve essere dichiarato tra parentesi angolari prima del nome del metodo. Però in questo caso è necessario che l'invocazione del metodo sia qualificata appropriatamente tramite il nome della classe per metodi statici e con `this` o `super` per metodi non statici. Ecco alcuni esempi:

```
        k = Test.<String>find(intA, "A");    // ERRORE in compilazione
```

```
        k = Test.<Integer>find(intA, "C");   // ERRORE in compilazione
        k = Test.<Integer>find(intA, 2);     // OK
```

In questo modo è come se avessimo definito tantissime versioni (non generiche) del metodo `find()`, una per il tipo `String`, una per il tipo `Integer`, e così via per ogni possibile tipo riferimento. Usato in questo modo il metodo generico permette di ottenere la massima generalità, come la versione che usa `Object`, mantenendo però il controllo statico del tipo. Così l'incongruità di una invocazione come `find(intA, "A")`, che cerca una stringa in un array di interi, viene rilevata durante la compilazione. Si osservi che questo tipo di errore potrebbe essere molto difficile da rilevare durante l'esecuzione perché non provoca nessun lancio di eccezioni.

Quindi la genericità permette di scrivere un'unica versione generica di un metodo che sta per tutte le versioni che potrebbero essere scritte per tutti i tipi riferimento che possono essere sostituiti alle variabili di tipo. Inoltre il codice prodotto dal compilatore non è inutilmente gonfiato perchè esiste un'unica versione compilata del metodo generico e questa è essenzialmente la stessa che sarebbe stata prodotta scrivendo il metodo con il tipo `Object` sostituito alle variabili di tipo (con l'aggiunta in certi casi di opportuni cast).

Consideriamo ora un'altro esempio: un metodo che preso in input un array e un valore assegna a tutte le componenti dell'array il valore dato. Anche in questo caso definiamo sia la versione generica che quella che usa il tipo `Object`.

```
public class Test {
   public static <T> void fill(T[] a, T x) {
        for (int i = 0 ; i < a.length ; i++)
            a[i] = x;
    }

    public static void fill2(Object[] a, Object x) {
        for (int i = 0 ; i < a.length ; i++)
            a[i] = x;
    }

    public static void main(String[] args) {
        String[] sA = ...
        Integer[] intA = ...
        fill(intA, 13);
        fill2(intA, 13);
        fill(intA, "A");                 // ERRORE in esecuzione: ArrayStoreException
        fill2(intA, "A");                // ERRORE in esecuzione: ArrayStoreException
        Test.<Integer>fill(intA, "A");   // ERRORE in compilazione
        fill(sA, "A");
        fill(sA, 12);                    // ERRORE in esecuzione: ArrayStoreException
        Test.<String>fill(sA, 12);       // ERRORE in compilazione
        Test.<String>fill(sA, "A");
    }
}
```

Una invocazione incongrua come `fill(intA, "A")` provoca un errore in esecuzione con lancio dell'eccezione `ArrayStoreException` perché si è tentato di assegnare un valore di tipo `String` a una componente di un array di `Integer`. Se si usa l'invocazione `Test.<Integer>fill(intA, "A")` l'errore è rilevato in compilazione. A differenza della versione `fill2()` del metodo per cui non c'è nessun modo per far sì che lo stesso errore sia rilevabile in compilazione.

Il prossimo esempio mostra che la genericità permette di evitare (o perlomeno limitare) l'uso di cast che potrebbero fallire in esecuzione. Il metodo `getMiddle()` semplicemente ritorna il valore che si trova nella posizione centrale dell'array di input. Il metodo `longestString()` ritorna il valore dell'array di input con la massima lunghezza della stringa data dal metodo `toString()`.

```
public class Test {
    public static <T> T getMiddle(T[] a) {
        return a[a.length/2];
    }

    public static Object getMiddle2(Object[] a) {
        return a[a.length/2];
    }

    public static <T> T longestString(T[] a) {
        T val = null;
```

```
        int max = 0;
        for (int i = 0 ; i < a.length ; i++)
            if (a.toString().length() >= max) val = a[i];
        return val;
    }

    public static void main(String[] args) {
        String[] sA = ...
        Integer[] intA = ...
        int k = getMiddle(intA);
        k = getMiddle(sA);              // ERRORE in compilazione
        String s = getMiddle(sA);
        s = getMiddle2(sA);             // ERRORE in compilazione: richiesto cast (String)
        s = (String)getMiddle2(sA);     // OK
        k = (Integer)getMiddle2(sA);    // ERRORE in esecuzione: ClassCastException

        k = longestString(intA);
        s = longestString(sA);
    }
}
```

Nel metodo `getMiddle()` la variabile di tipo determina anche il tipo del valore ritornato. Quindi il tipo inferito dipende sia dal tipo dell'argomento che dal tipo della variabile a cui il valore ritornato dal metodo deve essere assegnato. Inoltre, se il tipo inferito non è compatibile con il tipo della variabile a cui il valore ritornato deve essere assegnato, si produce un errore in compilazione. Ad esempio l'invocazione `k = getMiddle(sA)` provoca un errore in compilazione perché il tipo inferito, dopo le conversioni boxing/unboxing, è un supertipo stretto di `Integer` e che quindi non può essere assegnato ad un `Integer`. Si osservi inoltre che mentre per la versione generica l'invocazione lecita `s = getMiddle(sA)` si può scrivere senza cast, la stessa invocazione per la versione non generica `getMiddle2()` richiede un cast. Ma nel momento in cui si introduce un cast si corre il rischio che questo fallisca in esecuzione. Ad esempio, si potrebbe sostituire, inavertitamente, `sA` con `intA` nell'invocazione di `getMiddle2()` e questo non sarebbe rilevato in compilazione mentre produrrebbe poi un errore in esecuzione.

Si osservi che nel metodo `longestString()` la variabile di tipo è usata anche per definire il tipo di una variable locale. Come abbiamo già detto le variabili di tipo possono essere usate come se fossero il nome di un tipo effettivo. Però c'è una importante eccezione: le variabili di tipo non possono essere usate in espressioni creazionali come `new T(...)` o `new T[...]`. Inoltre, non si possono usare con l'operatore `instanceof`. Quindi in un metodo generico non si possono creare (direttamente) oggetti o array il cui tipo è quello di una variabile di tipo. Il prossimo esempio mostra proprio un caso del genere. Un metodo che preso in input un array ne crea una copia con gli elementi mescolati casualmente. Il nuovo array il cui tipo delle componenti deve essere lo stesso di quello di input, e quindi dato dalla variabile di tipo `T`, non può essere creato con `new T[...]`, però può essere creato tramite il metodo `clone()`.

```
public class Test {
    public static <T> T[] shuffledShallowCopy(T[] a) {
        T[] copy = a.clone();
        int n = copy.length;
        int m = 20*n;
        for (int i = 0 ; i < m ; i++) {
            int k = (int)Math.random()*n;
            int h = (int)Math.random()*n;
            T x = copy[k];
            copy[k] = copy[h];
            copy[h] = x;
        }
        return copy;
    }

    public static void main(String[] args) {
        String[] sA = ...
        Integer[] intA = ...
        String[] shuffledSA = shuffledShallowCopy(sA);
        Integer[] shuffledIntA = shuffledShallowCopy(intA);
        shuffledSA = shuffledShallowCopy(intA);          // ERRORE in compilazione
        Object[] objA;
        objA = shuffledShallowCopy(intA);                // OK
        intA = shuffledShallowCopy(objA);                // ERRORE in compilazione
    }
}
```

Questa tecnica di usare `clone()` non va bene se il nuovo array deve avere uan lunghezza maggiore rispetto a quello dell'array di input. In questi casi si possono usare altre tecniche che vedremo più avanti.
Negli esempi che abbiamo visto finora i metodi generici sono statici e una sola variabile di tipo è usata. In generale, come vedremo a breve, si possono definire metodi generici non statici e si possono usare due o più variabili di tipo. Per quanto riguarda i nomi delle variabili di tipo non ci sono restrizioni particolari, seguono le stesse regole di un qualsiasi altro nome di variabile. Però, per meglio distinguerle da nomi di variabili e nomi di tipi, è consuetudine che i loro nomi consistano in singole lettere maiuscole (`T`, `S`, ecc.).

Esercizi

[Errori_MG_1] Il seguente programma contiene uno o più errori. Trovare gli errori e spiegarli. In particolare, dire per ogni errore se si verifica in compilazione o durante l'esecuzione.

```
public class Test {
    public static <T> T first(T[] A, T[] B) {
        int i = 0;
        while (i < A.length && i < B.length && !A[i].equals(B[i]))
            i++;
        return (i < A.length && i < B.length ? A[i] : null);
    }
    public static void main(String[] args) {
        long[] longA = {2, new Long(5)};
        int[] intA = {1, 2, 3};
        int val = first(longA, longA);
        val = first(longA, intA);
        Integer[] intA2 = {2, 3, 4};
        Long[] longA2 = {1L, 2L, 3L};
        val = first(intA2, longA2);
        Long vL = first(intA2, longA2);
        Number num = first(intA2, longA2);
    }
}
```

[Errori_MG_2] Il seguente programma contiene uno o più errori. Trovare gli errori e spiegarli. In particolare, dire per ogni errore se si verifica in compilazione o durante l'esecuzione.

```
public class Test {
    public static <E> E sub(E[] a, E v) {
        int i = 0;
        while (i < a.length && !v.toString().equals(a[i].toString()))
            i++;
        if (i < a.length) {
            E c = a[i];
            a[i] = v;
            return c;
        } else return null;
    }
    public static void main(String[] args) {
        String[] sA = {"A", "B", "C"};
        String s = sub(sA, "D");
        Integer[] intA = {1, 2, 3};
        int k = sub(intA, 13);
        k = sub(intA, "2");
        Object obj = sub(intA, "3");
        obj = Test.<Integer>sub(intA, '4');
        Object[] objA = {2, 3, 4};
        Integer v = sub(objA, 2);
    }
}
```

[Elementi_comuni] Scrivere un metodo generico che presi in input due array ritorna il numero di elementi del primo array che sono presenti anche nel secondo array. Scrivere anche una versione non generica usando il tipo `Object`. Confrontare le due versioni e discutere i vantaggi e i svantaggi dei loro possibili usi.

[Valore_comune] Scrivere un metodo generico che presi in input due array ritorna il primo valore del primo array che appare anche nel secondo. Se non ci sono valori in comune ritorna `null`. Scrivere anche

una versione non generica del metodo e discutere i vantaggi e i svantaggi dei possibili usi delle due versioni.

[Stampa_matrici] Scrivere un metodo generico che presa in input una matrice la stampa in modo che le colonne siano allineate come nei seguenti esempi:

```
MATRICE di Double                    MATRICE di String
0.123    23.5    12.01               Roma    Milano           Genova
1        0.4     1.234               Napoli  Reggio Calabria  Teramo
1234567  67.897  12                  Terni   Palermo          Perugia
```

Scrivere anche una versione non generica e discutere i vantaggi e i svantaggi dei possibili usi delle due versioni.

[Sostituzione] Scrivere un metodo generico che prende in input un array e due valori `x` e `y`, e crea e ritorna una copia dell'array di input con tutti i valori `x` sostituiti dal valore `y`.

Tipi generici

Al pari dei metodi generici che aggiungono flessibilità al linguaggio senza sacrificare il controllo statico sui tipi, Java supporta anche la definizione di *tipi generici* (*generic types*). Un tipo generico è una classe o una interfaccia che nella sua dichiarazione ha una o più variabili di tipo (dichiarate tra parentesi angolari). Una variabile di tipo è usata nella definizione della classe/interfaccia alla stregua di un qualsiasi nome di tipo effettivo, con le stesse limitazioni viste per i metodi generici. Ogni tipo generico definisce un insieme di *tipi parametrici* (*parameterized types*) che consistono nel nome della classe o interfaccia seguito da una lista di nomi di tipi effettivi (tra parentesi angolari) corrispondenti alle variabili di tipo. Consideriamo un semplice esempio di tipo generico che rappresenta una coppia di valori:

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

    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() { return first; }
    public T getSecond() { return second; }

    public void setFirst(T x) { first = x; }
    public void setSecond(T x) { second = x; }
}
```

La variabile di tipo `T` è dichiarata, come per i metodi generici, tra parentesi angolari e immediatamente dopo il nome della classe (o dell'interfaccia). Poi, nella definizione della classe può essere usata come se fosse il nome di un tipo effettivo (eccetto che nelle espressioni creazionali). In generale, per istanziare un oggetto tramite un tipo generico è necessario specificare esplicitamente quali sono i tipi che devono essere sostituiti alle corrispondenti variabili di tipo. In altre parole, è necessario specificare quale dei tipi parametrici che possono corrispondere al tipo generico deve essere istanziato. Nel caso del tipo generico `Pair<T>` si deve specificare quale tipo effettivo (ad esempio, `String`, `Double`, ecc.) deve sostituire la variabile `T`. Ecco alcuni esempi di uso di questo tipo generico:

```
Pair<String> strPair = new Pair<String>("primo", "secondo");
strPair.setFirst("I");
strPair.setSecond(2);       // ERRORE in compilazione: Integer non è un sottotipo di String

Pair<Integer> intP = new Pair<Integer>(13, 17);    // boxing
intP.setFirst(11);          // boxing
intP.setSecond("13");       // ERRORE in compilazione: String non è un sottotipo di Integer
int k = intP.getFirst();    // unboxing
String s = intP.getFirst(); // ERRORE in compilazione: Integer non è un sottotipo di String

Pair<Pair<String>> strPairPair = new Pair<Pair<String>>(strPair, strPair);
```

Per istanziare una coppia di stringhe si usa il tipo parametrico `Pair<String>` e per istanziare una coppia di interi si usa `Pair<Integer>`. L'ultima riga mostra che un tipo parametrico può a sua volta essere usato

per creare un tipo parametrico (una coppia di coppie di stringhe). Una volta istanziato un tipo parametrico diventa del tutto simile a un tipo non generico definito direttamente con i tipi effettivi al posto delle variabili di tipo. Così il tipo `Pair<String>` è sostanzialmente indistinguibile dal seguente tipo `StrPair`, anche se i due tipi non sono equivalenti:

```
public class StrPair {
    private String first, second;

    public Pair(String first, String second) {
        this.first = first;
        this.second = second;
    }

    public String getFirst() { return first; }
    public String getSecond() { return second; }

    public void setFirst(String x) { first = x; }
    public void setSecond(String x) { second = x; }
}
```

Però il tipo (non generico) corrispondente ad un tipo parametrico non viene esplicitamente definito dal compilatore. Come per i metodi generici, il codice compilato contiene una sola classe che corrisponde al tipo generico e che è usata per tutti i tipi parametrici. Tale classe è molto simile a quella che si ottiene sostituendo la variabile di tipo con `Object`. Quindi il codice compilato non contiene nessuna indicazione del tipo effettivo che è stato usato per istanziare il tipo generico. Questo ha importanti conseguenze alcune delle quali saranno discusse ora e altre più avanti. Introduciamo una semplice classe e una sua sottoclasse che ci serviranno per esemplificare alcuni concetti.

```
// in un file Point.java
public class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }
}

// in un file LPoint.java
public class LPoint extends Point {
    private String label;

    public LPoint(int x, int y, String label) {
        super(x, y);
        this.label = label;
    }

    public String getLabel() { return label; }
}
```

Consideriamo ora il seguente frammento di programma:

```
Pair<Point> pntPair = new Pair<Point>(new Point(0, 0), new Point(1, 1));
pntPair.setFirst(new LPoint(0, 0, "A"));   // OK: LPoint è un sottotipo di Point

LPoint lp1 = new LPoint(0, 0, "A");
LPoint lp2 = new LPoint(1, 1, "B");
Pair<LPoint> lpPair = new Pair<LPoint>(lp1, lp2);
lpPair.setFirst(new Point(0, 0));   // ERRORE in compilazione: Point non è un
                                    // sottotipo di LPoint
pntPair = lpPair;                              // 1. ERRORE in compilazione
pntPair.setFirst(new Point(0, 0));             // 2. se (1) non fosse un errore allora...
String label = lpPair.getFirst().getLabel();   // 3. qui si avrebbe un errore in esecuzione

Pair<Object> objPair = new Pair<Object>("B", new Point(0, 0));
objPair.setFirst(lp1);
objPair = pntPair;   // ERRORE in compilazione: Pair<Point> non è un sottotipo di Pair<Object>
```

L'assegnamento `pntPair = lpPair` provoca un errore in compilazione perché il tipo di `lpPair`, che è `Pair<LPoint>`, non è un sottotipo del tipo di `pntPair`, che è `Pair<Point>`. Questo può apparire sorprendente ma se così non fosse allora sarebbero possibili le istruzioni (2) e (3). L'istruzione (3) provocherebbe un errore in esecuzione perchè il primo elemento della coppia in `lpPair` è stato posto dall'istruzione (2) uguale ad un oggetto di tipo `Point` il quale non ha il metodo `getLabel()`. Quindi se `Pair<LPoint>` fosse trattato come un sottotipo di `Pair<Point>` la genericità non potrebbe garantire la sua proprietà fondamentale: il controllo statico sui tipi. D'altronde se immaginiamo delle classi `PointPair` e `LPointPair` che definiscono direttamente coppie di `Point` e coppie di `LPoint` non c'è ragione perché debbano essere l'una il sottotipo dell'altra. In generale, quindi, se `Type1` e `Type2` sono due qualsiasi tipi distinti allora `Pair<Type1>` e `Pair<Type2>` non hanno nessuna relazione di sottotipo o supertipo fra loro. Questo è in contrasto con il comportamento degli array. Sappiamo che se `Type1` è un sottotipo di `Type2` allora `Type1[]` è un sottotipo di `Type2[]`. Ma sappiamo anche che gli array non permettono un controllo statico dei tipi:

```
LPoint[] lpA = new LPoint[3];
Point[] pA = lpA;              // lecito perché LPoint[] è un sottotipo di Point[]
pA[0] = new Point(0, 0);       // ERRORE in esecuzione (non rilevabile dal compilatore)
```

Infatti un array, a differenza dei tipi parametrici, mantiene nel codice compilato il tipo delle componenti. E deve essere così perchè altrimenti l'errore precedente non sarebbe rilevabile, neanche in esecuzione. Questa è la ragione per cui non è possibile creare array le cui componenti sono di un tipo parametrico:

```
Pair<String>[] strPairA;           // OK
strPairA = new Pair<String>[3];    // ERRORE in compilazione
```

Però è possibile dichiarare variabili il cui tipo è un array di un tipo parametrico. Discuteremo più avanti il perché di questa stranezza.

La classe generica `Pair` ha una sola variabile di tipo. Diamo ora un esempio che usa due variabili di tipo. Si tratta di una versione più generale della classe `Pair<T>` perché permette che i tipi delle due componenti possano essere diversi.

```
public class DPair<F, S> {
    private F first;
    private S second;

    public DPair(F first, S second) {
        this.first = first;
        this.second = second;
    }

    public F getFirst() { return first; }
    public S getSecond() { return second; }

    public void setFirst(F x) { first = x; }
    public void setSecond(S x) { second = x; }
}
```

Avremmo potuto definire prima questa classe generica e poi `Pair<T>` come sottoclasse (generica):

```
public class Pair<T> extends DPair<T, T> {
    public Pair(T first, T second) {
        super(first, second);
    }
}
```

Se avessimo fatto così, per ogni tipo specifico `Type`, si avrebbe che

`Pair<Type>` è un sottotipo di `DPair<Type, Type>`.

D'altronde ciò è in accordo con l'intuizione che equipara `Pair<Type>` ad una classe non generica che è definita estendendo una classe (non generica) che corrisponde a `DPair<Type, Type>`. Ovviamente questo vale in generale.

Interfacce generiche Non solo le classi ma anche le interfacce possono essere generiche. È anzi più facile incontrare esempi di interfacce generiche che di classi generiche e di solito le classi generiche sono implementazioni di interfacce generiche. Per le interfacce valgono le stesse regole di dichiarazione e uso delle variabili di tipo. Iniziamo considerando una delle interfacce generiche più implementate e usate della libreria di Java. L'interfaccia `Comparable<T>`, del package `java.lang`, serve ad imporre un

ordinamento totale degli oggetti della classe che la implementa:

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

Il metodo `compareTo(T obj)` deve ritornare un intero negativo se l'oggetto su cui è invocato è minore di `obj`, un intero positivo se invece è maggiore di `obj` e zero se sono uguali. Tantissime classi della libreria Java implementano questa interfaccia. Ad esempio, le otto classi che corrispondono ai tipi primitivi, `String`, `Date`, ecc. Ovviamente, le classi implementano un opportuno tipo parametrico (o interfaccia parametrica) che deriva dall'interfaccia generica `Comparable<T>`. La classe `String` implementa `Comparable<String>`, la classe `Integer` implementa `Comparable<Integer>`, `Double` implementa `Comparable<Double>` e così via.
Modifichiamo la definizione della classe `Point` implementando l'interfaccia `Comparable<Point>`:

```
public class Point implements Comparable<Point> {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    public int compareTo(Point p) {
        if (p == null) throw new NullPointerException();
        int px = p.getX(), py = p.getY();
        if (px == x && py == y) return 0;                // this è uguale a p
        else if (px < x || (px == x && py < y))
            return 1;                                    // this è maggiore di p
        else return -1;                                  // this è minore di p
    }
}
```

L'ordinamento implementato è molto semplice: ordina rispetto alle ascisse e a parità di ascissa ordina rispetto alle ordinate.
L'interfaccia generica `Comparable<T>` permette di implementare metodi generici che si basano su una relazione di ordinamento. Ad esempio, trovare il minimo di un array o un insieme, trovare il massimo, ordinare un array, cercare un valore in un array tramite ricerca binaria, ecc. Proviamo allora a definire un metodo generico che ritorna il valore minimo di un array:

```
public static <T> T min(T[] a) {
    T min = a[0];
    for (int i = 1 ; i < a.length ; i++)
        if (a[i].compareTo(min) < 0)    // ERRORE in compilazione: non trova compareTo()
            min = a[i];
    return min;
}
```

Questa implementazione non va bene perché il metodo `compareTo()` non è un metodo come quelli di `Object` che appartengono a tutti i tipi riferimento e quindi a tutti i possibili tipi che la variabile `T` può rappresentare. Affinché si possa scrivere un metodo generico come questo occorre che la variabile di tipo sia limitata ai tipi `T` che implementano l'interfaccia `Comparable<T>`. Ciò è possibile:

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

La dichiarazione `<T extends Comparable<T>>` significa proprio che la variabile di tipo `T` varia solamente tra i tipi che sono sottotipi di `Comparable<T>`, cioè, i tipi che implementano tale interfaccia. In altre parole, un tipo effettivo `Type` può sostituire la variabile di tipo `T` se e solo se `Type` implementa l'interfaccia `Comparable<Type>`. Per chiarire bene questo punto consideriamo un frammento di codice

che usa il metodo `min()`:

```
Point[] pA = new Point[] {new Point(2, 3), new Point(1, 5), new Point(1, 4)};
Point minP = min(pA);
System.out.println("Min Point = ("+minP.getX()+", "+minP.getY()+")");
Integer[] pI = new Integer[] {3, 4, 1, 0, -3, 6};
int m = min(pI);
System.out.println("Min Integer = "+m);
LPoint[] lpA = new LPoint[] {new LPoint(0, 0, "a"), new LPoint(0, -1, "b")};
LPoint minLP = min(lpA);       // ERRORE in compilazione
```

L'ultima riga provoca un errore in compilazione perchè la classe `LPoint` non implementa l'interfaccia `Comparable<LPoint>`. Però, siccome `LPoint` estende `Point`, la classe `LPoint` implementa l'interfaccia `Comparable<Point>`. E potrebbe essere che l'ordinamento inteso per gli oggetti di tipo `LPoint` sia proprio quello ereditato dalla superclasse `Point`. Per casi come questo si vorrebbe poter scrivere un metodo generico che è usabile anche quando l'interfaccia `Comparable` è implementata da una superclasse o più in generale, quando la classe implementa l'interfaccia `Comparable` relativamente a un supertipo. Vedremo presto che questo è invero possibile.
Troveremo molti altri esempi di interfacce e classi generiche quando tratteremo le collezioni della libreria Java.

Esercizi

[Errori_TG_1] Il seguente programma contiene uno o più errori. Trovare gli errori e spiegarli. In particolare, dire per ogni errore se si verifica in compilazione o durante l'esecuzione.

```
class LDPair<F, S> extends DPair<F, S> {
    private String label;
    public LDPair(F f, S s, String l) {
        super(f, s);
        label = l;
    }
    public String getLabel() { return label; }
}

public class Test {
    public static void main(String[] args) {
        DPair<Integer, Number> dp = new LDPair<Integer, Number>(12, 2.6, "A");
        String s = dp.getLabel();
        DPair<Object, Object> objp = new DPair<String, String>("A", "B");
        DPair<Long, Long>[] pA = new DPair<Long, Long>[5];
        DPair<DPair<Long, Long>, int[]> t =
                new DPair<DPair<Long, Long>, int[]>(new DPair<Long, Long>(1L, 1L), new int[5]);
    }
}
```

[Errori_TG_2] Il seguente programma contiene uno o più errori. Trovare gli errori e spiegarli. In particolare, dire per ogni errore se si verifica in compilazione o durante l'esecuzione.

```
class CN implements Comparable<Integer> {
    private String val;
    public CN(String v) { val = v; }

    public int compareTo(Integer o) {
        if (o == null) throw new NullPointerException();
        int len = val.length();
        return (len < o ? -1 : (len > o ? 1 : 0));
    }
}

public class Test {
    public static <T extends Comparable<Integer>> boolean below(T[] a, int b) {
        for (int i = 0 ; i < a.length ; i++)
            if (a[i].compareTo(b) > 0) return false;
        return true;
    }
    public static void main(String[] args) {
        CN[] a = new CN[] {new CN("A"), new CN("B")};
```

```
        boolean r = below(a, 13);
        Integer[] intA = new Integer[] {1, 2, 3};
        r = below(intA, 10);
        int[] iA = {1, 2, 3};
        r = below(iA, 5);
    }
}
```

[Punti_generici] Definire una versione generica `GLPoint<T>` della classe `LPoint` in cui la `label` ha tipo generico `T`. La classe `GLPoint<T>` deve estendere la classe `Point`. Così `LPoint` corrisponderebbe a `GLPoint<String>`.
[Copia_e_scambia] Definire un metodo generico `copySwap` che presa in input una coppia di tipo `Pair<T>`, crea e ritorna una nuova coppia dello stesso tipo con i valori della coppia di input scambiati.
[Minmax] Scrivere un metodo generico che preso in input un array ritorna il valore minimo e il valore massimo dell'array. Usare `Pair<T>` per ritornare la coppia di valori.
[Conta_valore] Scrivere un metodo generico che preso in input un array ritorna il valore più frequente e il numero di occorrenze (usare in modo opportuno il tipo `DPair<F, S>` per ritornare la coppia valore e numero occorrenze). Ecco alcuni esempi di input e output del metodo:

```
INPUT                                   OUTPUT
{"B", "AB", "A", "B"}                   ("B", 2)
{2, 13, 2, 1, 13, 13}                   (13, 3)
```

[Insiemi_generici] Definire una classe `GSet<T>` per rappresentare insiemi i cui elementi sono di tipo (generico) `T`. Implementare dei metodi per le seguenti operazioni:

- determinare se un dato oggetto è presente o meno nell'insieme;
- aggiungere un oggetto all'insieme (se non è già presente);
- rimuovere un oggetto dall'insieme.

[Multinsieme] Definire una classe generica `MSet<T>` per rappresentare multi-insiemi i cui elementi sono di tipo (generico) `T`. Un multi-insieme a differenza di un insieme può contenere uno stesso elemento più volte. In altri termini ogni elemento appartenente al multi-insieme vi appartiene con una certa molteplicità (1, 2, 3, ...). Implementare dei metodi per le seguenti operazioni:

- determinare la molteplicità di un dato oggetto (se non è presente, la molteplicità è zero);
- aggiungere un oggetto al multi-insieme (se è già presente, la molteplicità è incrementata di uno);
- rimuovere un oggetto dal multi-insieme (se è presente, la molteplicità è decrementata di uno).

Suggerimento: rappresentare ogni elemento con una coppia (valore, molteplicità).
[Conta_valori] Scrivere un metodo generico che preso in input un array ritorna un `MSet<T>` (vedi l'esercizio [Multinsieme]) che rappresenta i valori presenti nell'array. Ecco un esempio:

```
ARRAY          {"A", "AB", "B", "A", "AB", "AB"}
MULTI-INSIEME  {("A", 2), ("AB", 3), ("B", 1)}
```

[Code_generiche] Definire una interfaccia generica per code e fornire delle implementazioni (tramite classi generiche) con liste e con array.

Wildcards

Sappiamo che i tipi generici e più precisamente i tipi parametrici si comportano in modo diverso dagli array per quanto riguarda le relazioni di sottotipo e, simmetricamente, di supertipo. Ricordiamo che se `TypeB` è un sottotipo di `TypeA` allora `TypeB[]` è un sottotipo di `TypeA[]`, mentre `GenType<TypeB>` **non** è un sottotipo di `GenType<TypeA>` (a meno che `TypeA` non coincida con `TypeB`), dove `GenType<T>` è un qualsiasi tipo generico. Ciò è necessario affinché si possa garantire un controllo statico sui tipi parametrici. Però in alcune situazioni, come nel caso del precedente metodo `min()`, sarebbe utile avere un modo, dato un tipo `Type`, per riferirsi a un qualsiasi tipo parametrico `GenType<SubT>` tale che `SubT` è un sottotipo di `Type`, o anche a un qualsiasi tipo parametrico `GenType<SuperT>` tale che `SuperT` è un supertipo di `Type`. Il linguaggio Java permette di fare ciò con la sintassi delle *wildcards*:

`<? extends Type>`
rappresenta un qualsiasi tipo che è un sottotipo di `Type`.

`<? super Type>`
: rappresenta un qualsiasi tipo che è un supertipo di `Type`.

`<?>`
: rappresenta un qualsiasi tipo che è un sottotipo di `Object`, quindi un qualsiasi tipo riferimento senza restrizioni. È equivalente a `<? extends Object>`.

La wildcard è rappresentata dal carattere `?` e `Type` può essere un tipo effettivo, una variabile di tipo o un tipo generico. La wildcard non è una variabile di tipo perché il suo *scope* è ristretto alle parentesi angolari in cui è usata. Le espressioni con wildcards possono essere usate ovunque si può usare il nome di un tipo, con alcune eccezioni che discuteremo fra poco.

Per alcuni esempi ci sarà utile la seguente classe generica che rappresenta array dinamici, cioè, array la cui lunghezza può essere variata a piacimento:

```
import static java.util.Arrays.*;

public class DynamicArray<T> {
    private T[] array;

    public DynamicArray() {
        array = (T[])new Object[0];
    }

    public int getSize() { return array.length; }

    public void setSize(int size) {
        if (size < 0) throw new IllegalArgumentException();
        array = copyOf(array, size);
    }

    public T get(int index) {
        if (index < 0 || index >= array.length)
            throw new IllegalArgumentException();
        return array[index];
    }

    public void set(int index, T x) {
        if (index < 0 || index >= array.length)
            throw new IllegalArgumentException();
        array[index] = x;
    }
}
```

Se ci occorresse un metodo generico che stampa un array dinamico potremmo scriverlo così:

```
public static <T> void printDynArray(DynamicArray<T> a) {
    int size = a.getSize();
    for (int i = 0 ; i < size ; i++) System.out.println(a.get(i));
}
```

Però la variabile di tipo `T` è usata solamente per denotare il tipo dell'array dinamico. In questi casi può essere sostituita con una wildcard:

```
public static void printDynArray(DynamicArray<?> a) {
    int size = a.getSize();
    for (int i = 0 ; i < size ; i++) System.out.println(a.get(i));
}
```

Questa versione è del tutto equivalente a quella che usa la variabile di tipo. In tutti i casi, come questo, in cui una variabile di tipo è usata una sola volta (cioè, non ci sono dipendenze legate a tale variabile) si preferisce sostituirla con una wildcard.

Supponiamo di voler aggiungere alla classe `DynamicArray<T>` un metodo che prende in input un array dinamico e ne appende i valori all'array dinamico dell'oggetto. Potremmo definirlo così:

```
    public void append(DynamicArray<T> a) {
        int size = a.getSize();
        int oldSize = array.length;
        setSize(oldSize + size);
        for (int i = 0 ; i < size ; i++) array[oldSize + i] = a.get(i);
    }
```

Però, potrebbe essere troppo restrittivo:

```
DynamicArray<Point> pntDA = new DynamicArray<Point>();
DynamicArray<LPoint> lpDA = new DynamicArray<LPoint>();
...
pntDA.append(lpDA);    // ERRORE in compilazione
```

L'errore è dovuto al fatto che `DynamicArray<LPoint>` non è un sottotipo di `DynamicArray<Point>`. D'altronde non c'è ragione di impedire che l'array dinamico di `Point` non possa anche contenere elementi di un suo sottotipo come lo è `LPoint`. Quindi potremmo riscrivere l'intestazione del metodo per permettere che si possa appendere un array dinamico di un sottotipo:

```
    public <E extends T> void append(DynamicArray<E> a) {
        ...
    }
```

Ora è lecito fare l'invocazione `pntDA.append(lpDA)`. Però la variabile di tipo `E` è usata una sola volta quindi è preferibile sostituirla con una wildcard:

```
    public void append(DynamicArray<? extends T> a) {
        ...
    }
```

Così è più semplice e più leggibile. Ora vogliamo aggiungere alla classe un metodo che copia i valori dell'array dell'oggetto in un dato array dinamico. Potremmo implementare tale metodo nel seguente modo:

```
    public void copyTo(DynamicArray<T> a) {
        int size = a.getSize();
        int length = array.length;
        if (size < length) a.setSize(length);
        for (int i = 0 ; i < length ; i++)
            a.set(i, array[i]);
    }
```

Così definito può essere troppo restrittivo:

```
DynamicArray<Point> pntDA = new DynamicArray<Point>();
DynamicArray<Point> pntDA2 = new DynamicArray<Point>();
DynamicArray<LPoint> lpDA = new DynamicArray<LPoint>();
...
pntDA.copyTo(pntDA2);     // OK
lpDA.copyTo(pntDA);       // ERRORE in compilazione
```

L'errore è dovuto al fatto che `DynamicArray<Point>` non è un sottotipo di `DynamicArray<LPoint>`. Però in questo caso avrebbe senso poter copiare elementi di tipo `LPoint` in un array dinamico di `Point`, essendo `Point` un supertipo di `LPoint`. Per ovviare a ciò possiamo rendere più flessibile il metodo sostituendo la variabile di tipo `T` con l'espressione che sta per un qualsiasi tipo che è un supertipo di `T`:

```
    public void copyTo(DynamicArray<? super T> a) {
        ...
    }
```

Con questa versione è lecito fare invocazioni come `lpDA.copyTo(pntDA)`, perché ora il metodo accetta in input un qualsiasi array dinamico di un supertipo di `T`. È importante notare che, a differenza della limitazione `extends` che può essere applicata anche a variabili di tipo, la limitazione `super` può essere usata solamente con una wildcard. Grazie a questo uso della wildcard possiamo riscrivere il metodo generico `min()`:

```
public static <T extends Comparable<? super T>> T min(T[] a) {
    ...
}
```

Con questa versione è ora possibile invocare il metodo con un array di un tipo che implementa l'interfaccia `Comparable` relativamente ad un supertipo, come nel caso di un array di `LPoint`:

```
LPoint[] lpA = ...
LPoint minLP = min(lpA);
```

Vediamo ora altri esempi per chiarire meglio il significato e l'uso delle wildcards. Consideriamo quattro versioni diverse di un metodo generico che copia un array dinamico in un'altro:

```
public class Test {
    public static <T> void copy(DynamicArray<? super T> dst, DynamicArray<? extends T> src) {
        int size = src.getSize();
        if (dst.getSize() < size) dst.setSize(size);
        for (int i = 0 ; i < size ; i++)
            dst.set(i, src.get(i));
    }
    public static <T> void copy2(DynamicArray<T> dst, DynamicArray<? extends T> src) {
        // omessa perché uguale a quella del metodo copy()
    }
    public static <T> void copy3(DynamicArray<? super T> dst, DynamicArray<T> src) {
        // omessa perché uguale a quella del metodo copy()
    }
    public static <T> void copy4(DynamicArray<T> dst, DynamicArray<T> src) {
        // omessa perché uguale a quella del metodo copy()
    }
}
```

Il seguente frammento di codice mostra le differenze tra le quattro versioni:

```
DynamicArray<Object> objDA = new DynamicArray<Object>();
DynamicArray<Number> numDA = new DynamicArray<Number>();
DynamicArray<Integer> intDA = new DynamicArray<Integer>();
...
copy(objDA, intDA);        // T è inferito essere Integer
copy2(objDA, intDA);       // T è inferito essere Object
copy3(objDA, intDA);       // T è inferito essere Integer
copy4(objDA, intDA);       // ERRORE in compilazione

copy(intDA, objDA);        // ERRORE in compilazione

Test.<Object>copy(objDA, intDA);
Test.<Number>copy(objDA, intDA);
Test.<Integer>copy(objDA, intDA);

Test.<Object>copy2(objDA, intDA);
Test.<Number>copy2(objDA, intDA);   // ERRORE in compilazione
Test.<Integer>copy2(objDA, intDA);  // ERRORE in compilazione

Test.<Object>copy3(objDA, intDA);   // ERRORE in compilazione
Test.<Number>copy3(objDA, intDA);   // ERRORE in compilazione
Test.<Integer>copy3(objDA, intDA);

Test.<Object>copy4(objDA, intDA);   // ERRORE in compilazione
Test.<Number>copy4(objDA, intDA);   // ERRORE in compilazione
Test.<Integer>copy4(objDA, intDA);  // ERRORE in compilazione
```

Il metodo `copy()` è il più flessibile dei quattro mentre `copy4()` è il più restrittivo. Le versioni `copy()`, `copy2()` e `copy3()` ammettono le stesse invocazioni quando il parametro di tipo è implicito (può variare solamente il tipo che è inferito per la variabile di tipo `T`). Quando invece il parametro di tipo è dichiarato esplicitamente anche le prime tre versioni si differenziano.

Consideriamo ora un metodo che calcola la somma dei valori di un array dinamico di numeri:

```
public class Test {
    public static double sum(DynamicArray<? extends Number> a) {
        double sum = 0;
        int n = a.getSize();
        for (int i = 0 ; i < n ; i++)
            sum += a.get(i).doubleValue();    // unboxing
        return sum;
    }
}
```

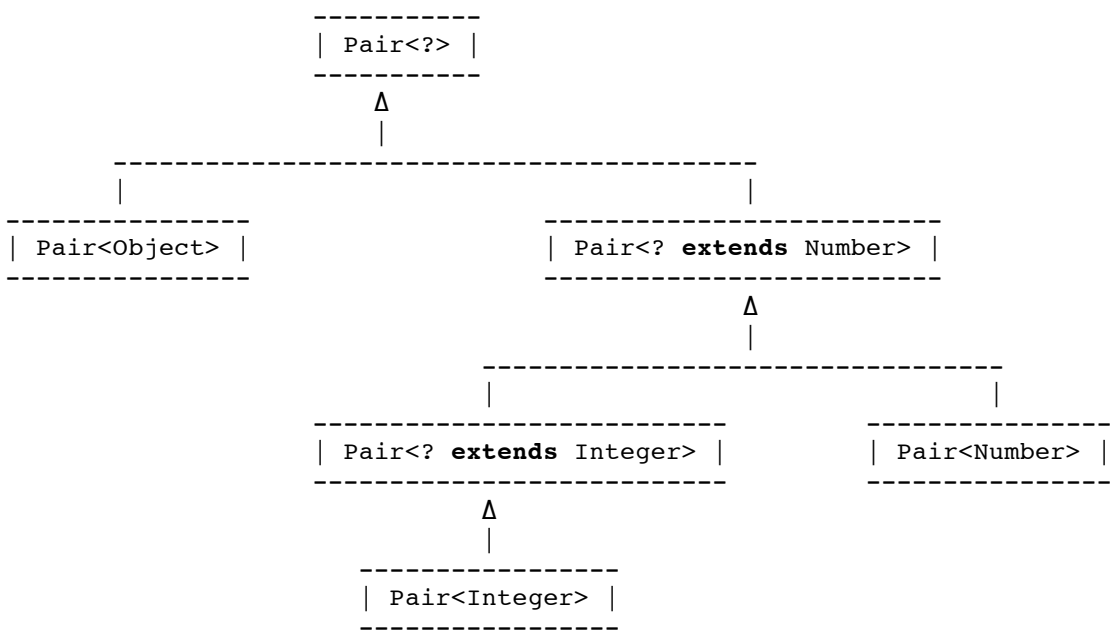
Ed ecco come il metodo può essere usato:

```
DynamicArray<Integer> ints = new DynamicArray<Integer>();
DynamicArray<Float> floats = new DynamicArray<Float>();
DynamicArray<Number> nums = new DynamicArray<Number>();
```

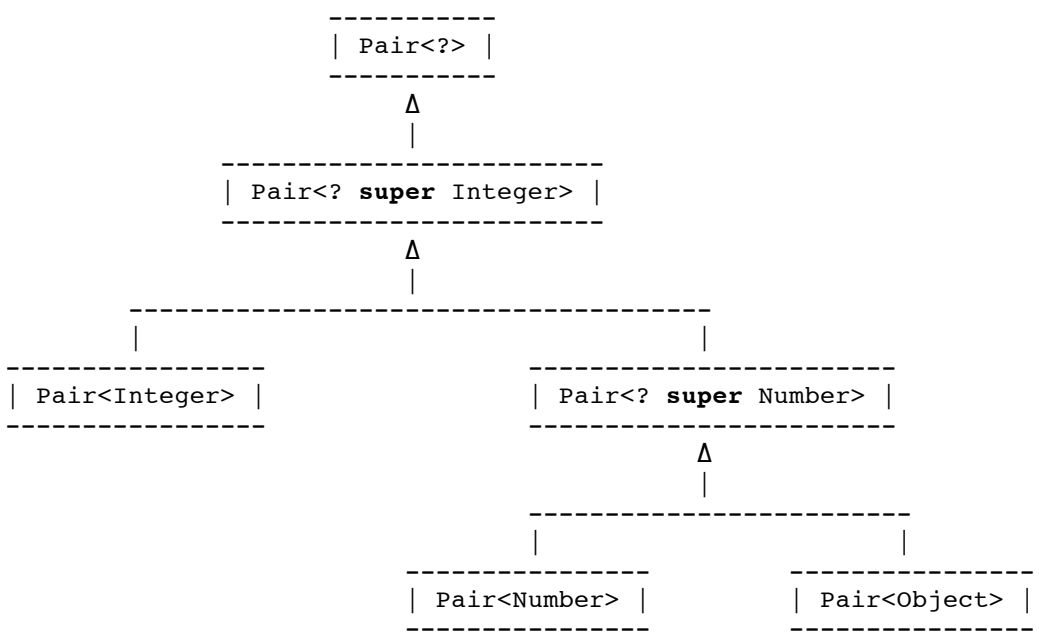
```
DynamicArray<String> strs = new DynamicArray<String>();
...
double tot = sum(ints);
tot = sum(floats);
tot = sum(nums);
tot = sum(strs);       // ERRORE in compilazione
```

Quindi il metodo `sum()` accetta come input solamente gli array dinamici di tipi numerici, cioè, sottotipi di `Number`.

Le espressioni che usano wildcard rappresentano dei tipi che sono chiamati *tipi wildcard* (*wildcard types*). I tipi wildcard hanno relazioni di sottotipo/supertipo tra loro e con i tipi parametrici ordinari. Il seguente diagramma mostra alcuni esempi di queste relazioni per wildcard con la limitazione **`extends`**:



Relazioni simmetriche valgono anche per wildcard con la limitazione **`super`**:



Vediamo ora alcuni esempi che mostrano le conseguenze di tali relazioni nella dichiarazione di variabili e nell'assegnamento di valori a variabili:

```
DynamicArray<?> da = new DynamicArray<?>();        // ERRORE in compilazione

DynamicArray<Pair<?>> pairDA = new DynamicArray<Pair<?>>();
pairDA.setSize(10);
pairDA.set(0, new Pair<Integer>(1, 2));
pairDA.set(1, new Pair<String>("a", "b"));
Pair<Integer> intPair = null;
```

```
intPair = pairDA.get(0);           // ERRORE in compilazione
Pair<Object> objPair = null;
objPair = pairDA.get(1);           // ERRORE in compilazione
Pair<?> unknown = pairDA.get(0);

DynamicArray<Pair<? extends Number>> numPairDA = new DynamicArray<Pair<? extends Number>>();
numPairDA.setSize(10);
numPairDA.set(0, new Pair<Integer>(1, 2));
numPairDA.set(1, new Pair<String>("a", "b"));    // ERRORE in compilazione
intPair = numPairDA.get(0);        // ERRORE in compilazione
Pair<Number> numPair = null;
numPair = numPairDA.get(0);        // ERRORE in compilazione

Pair<? extends Number> extNumPair = numPairDA.get(0);

extNumPair = intPair;
Pair<? super Integer> supIntPair = numPair;
supIntPair = extnumPair;           // ERRORE in compilazione
Pair<? super Number> supNumPair = null;
supIntPair = supNumPair;
supNumPair = supIntPair;           // ERRORE in compilazione
unknown = supIntPair;
unknown = extNumPair;
supIntPair = objPair;
```

La prima riga mostra che i tipi wildcard non possono essere usati in espressioni creazionali però, come mostra la seconda riga, possono essere usati se sono nidificati. Tutti gli altri errori derivano dalla non validità di relazioni di sottotipo.

Esercizi

[Errori_W_1] Il seguente codice Java contiene uno o più errori. Trovare gli errori e spiegarli. In particolare, dire per ogni errore se si verifica in compilazione o durante l'esecuzione.

```
public class Test {
    public static void main(String[] args) {
        DynamicArray<String> strDA = new DynamicArray<String>();
        strDA.setSize(10);
        strDA.set(0, "a");
        DynamicArray<?> da = strDA;
        da.set(0, "b");
        Object o = da.get(0);
        String s = da.get(0);
        DynamicArray<? extends String> sDA = strDA;
        sDA.set(0, "c");
        String s2 = sDA.get(0);
    }
}
```

[Errori_W_2] Il seguente codice Java contiene uno o più errori. Trovare gli errori e spiegarli. In particolare, dire per ogni errore se si verifica in compilazione o durante l'esecuzione.

```
public class Test {
    public static void main(String[] args) {
        DynamicArray<Integer> intDA = new DynamicArray<Integer>();
        intDA.setSize(10);
        intDA.set(0, 12);
        DynamicArray<? super Integer> supIDA = intDA;
        supIDA.set(1, 17);
        Integer intO = supIDA.get(0);
        DynamicArray<?> da = new DynamicArray<? extends Integer>();
        DynamicArray<?> da2 = new DynamicArray<DynamicArray<?>>();
    }
}
```

[Componenti_comuni] Aggiungere un metodo alla classe `DynamicArray<T>` che preso in input un array dinamico ritorna il numero di componenti dell'array dinamico oggetto che sono uguali a elementi dell'array dinamico di input. Il metodo deve essere flessibile. Se ad esempio l'array dinamico dell'oggetto

su cui è invocato ha il tipo delle componenti che è `Point`, allora il metodo deve accettare array dinamici con tipo uguale a un qualsiasi sottotipo di `Point`, come `LPoint`.
[Elementi_differenti] Aggiungere alla classe `DynamicArray<T>` un metodo che preso in input un array dinamico appende ad esso tutti gli elementi dell'array dinamico dell'oggetto che non sono già presenti nell'array di input. Il metodo deve essere flessibile. Se ad esempio l'array dinamico dell'oggetto su cui il metodo è invocato ha il tipo degli elementi che è `LPoint`, allora il metodo deve accettare array dinamici con tipo uguale a un qualsiasi supertipo di `LPoint`, come `Point`.
[Intersezione] Aggiungere alla classe `DynamicArray<T>` un metodo che preso in input un array dinamico crea e ritorna un nuovo array dinamico che contiene tutti gli elementi dell'array dinamico dell'oggetto che sono uguali a elementi presenti nell'array di input. Come deve essere definito per avere la massima flessibilità compatibilmente con un uso appropriato?
[Unione] Definire un metodo generico che prende in input due array dinamici e crea e ritorna un nuovo array dinamico che contiene l'unione degli elementi dei due array. Il metodo dovrebbe essere il più flessibile possibile.
[Max] Definire un metodo generico che preso in input un array dinamico ritorna il valore massimo dell'array. Come al solito, massimizzare la flessibilità del metodo.

Collezioni

Nella programmazione si presenta frequentemente la necessità di gestire un insieme, una lista, una coda, ovvero, un qualche tipo di collezione di oggetti. Proprio per venire incontro a questa diffusa esigenza la piattaforma Java mette a disposizione dei programmatori il *framework* denominato *`Collections`*, nel package `java.util`. Si tratta di un insieme ben organizzato di interfacce, classi astratte e classi concrete che, anche grazie alla genericità, fornisce un valido aiuto in tutti i casi in cui c'è bisogno di usare un qualche tipo di collezione di oggetti.

`Iterable` e *for-each*

Una interfaccia molto importante per quasi tutte le collezioni è `Iterable`, nel package `java.lang`:

```
public interface Iterable<E> {
        // ritorna un iteratore per elementi di tipo E
    Iterator<E> iterator();
}
```

Scopo dell'interfaccia `Iterable` è di stabilire una modalità standard per visitare (o scorrere) tutti gli elementi di una collezione. L'unico metodo dell'interfaccia, `iterator()`, ritorna un riferimento ad un oggetto di tipo `Iterator<E>`. A sua volta `Iterator<E>` è una interfaccia generica definita nel package `java.util`:

```
public interface Iterator<E> {
        // ritorna true se ci sono ancora elementi da visitare
    boolean hasNext();
        // ritorna il prossimo elemento
    E next();
        // rimuove l'elemento ritornato dall'ultima invocazione di next()
    void remove();
}
```

Il metodo `next()`, se invocato quando non ci sono più elementi da visitare, lancia un'eccezione di tipo `NoSuchElementException`. Il metodo `remove()`, se invocato quando `next()` non è stato ancora invocato o `remove()` è stato già invocato dopo l'ultima invocazione di `next()`, lancia un'eccezione di tipo `IllegalStateException`. Inoltre, il metodo `remove()` è facoltativo, nel senso che una classe che implementa l'interfaccia `Iterator` può anche non supportare il metodo (ad esempio, quando la collezione non è modificabile). Se `remove()` non è supportato allora l'invocazione del metodo lancia un'eccezione di tipo `UnsupportedOperationException`.

Conosciamo già una classe che implementa l'interfaccia `Iterator` ed è la classe `Scanner` che implementa `Iterator<String>`. Infatti, i metodi `next()` e `hasNext()` di `Scanner` implementano proprio i metodi dell'interfaccia. Per ovvie ragioni il metodo `remove()` non è invece supportato. Una classe che

gestisce un qualche tipo di collezione implementa, generalmente, l'interfaccia più flessibile `Iterable`, come fanno quasi tutte le classi del framework Collections. La ragione della maggiore flessibilità dell'interfaccia `Iterable` sta nel permettere di usare più iteratori (cioè, oggetti di tipo `Iterator`) simultaneamente. Così è possibile fare due (o più) scansioni nidificate ed indipendenti della collezione, una con un iteratore e l'altra con un'altro iteratore. Ciò è necessario se, ad esempio, si vogliono confrontare gli elementi a due a due. Questa flessibilità in più è anche il motivo per cui la classe `Scanner` non implementa l'interfaccia `Iterable`.

Implementazione tramite classe nidificata statica Prima di vedere le principali classi del framework Collections e quindi anche molti esempi di classi che implementano l'interfaccia `Iterable`, vediamo come l'interfaccia può essere implementata e usata. Vogliamo quindi definire una semplice classe per gestire collezioni di elementi che implementa l'interfaccia `Iterable`. Per implementare `Iterable` useremo una classe nidificata statica che implementa l'interfaccia `Iterator`, così che un'istanza di tale classe realizza un iteratore che può essere ritornato dal metodo `iterator()`.

```
import java.util.*;

public class ACollection<E> implements Iterable<E> {
    private E[] array;          //Array che mantiene gli elementi della collezione
    private int size;           //Numero di elementi presenti nella collezione
    private int modCount = 0;  //Contatore delle modifiche (serve all'iteratore)
        //Costruisce una collezione vuota
    public ACollection() {
        array = (E[]) new Object[0];
        size = 0;
    }
        //Metodo ausiliario che ritorna il primo indice dell'array che contiene
    private int find(E x) {       //l'elemento x, se non è presente ritorna -1.
        for (int i = 0 ; i < array.length ; i++)
            if (x == null ? array[i] == null : array[i].equals(x))
                return i;
        return -1;
    }
        //Metodo ausiliario che rimuove l'elemento in posizione i
    private void remove(int i) {
        array[i] = array[--size];
        modCount++;
    }
        //Ritorna true se l'elemento x è contenuto nella collezione
    public boolean contains(E x) {
        return (find(x) != -1);
    }
        //Aggiunge l'elemento x alla collezione (anche se è già presente)
    public void add(E x) {
        if (array.length == size)
            array = Arrays.copyOf(array, (3*size)/2 + 1);
        array[size++] = x;
        modCount++;
    }
        //Aggiunge alla collezione tutti gli elementi passati come argomenti
    public void addAll(E...a) {
        for (int i = 0 ; i < a.length ; i++) add(x);
    }
        //Rimuove dalla collezione la prima occorrenza dell'elemento x e ritorna true,
    public boolean remove(E x) {        //se non è presente ritorna false.
        int i = find(x);
        if (i != -1) {
            remove(i);
            return true;
        } else return false;
    }
        //Ritorna il numero di elementi della collezione
    public int size() { return size; }

        //Classe nidificata statica che implementa l'iteratore
    private static class Itr<T> implements Iterator<T> {
        private ACollection<T> coll;    //La collezione da iterare
        private int cursor = 0;         //Indice del prossimo elemento
        private int lastRet = -1;       //Indice dell'ultimo elemento ritornato, se non c'è -1.
```

```
        private int expectedModCount;   //Il conteggio atteso delle modifiche
            //Costruisce un iteratore per la collezione c
        private Itr(ACollection<T> c) {
            coll = c;
            expectedModCount = coll.modCount;    //Registra il conteggio attuale delle modifiche
        }
            //Controlla che non siano state fatte delle modifiche dall'esterno alla collezione
        private void checkForModification() {          //da quando l'iteratore è stato creato.
            if (expectedModCount != coll.modCount)
                throw new ConcurrentModificationException();
        }
            //Implementa il metodo dell'interfaccia Iterator
        public boolean hasNext() {
            checkForModification();
            return (cursor < coll.size);
        }
            //Implementa il metodo dell'interfaccia Iterator
        public T next() {
            if (!hasNext())
                throw new NoSuchElementException();
            lastRet = cursor;
            return coll.array[cursor++];
        }
            //Implementa il metodo dell'interfaccia Iterator
        public void remove() {
            checkForModification();
            if (lastRet == -1)
                throw new IllegalStateException();
            coll.remove(lastRet);
            cursor = lastRet;
            lastRet = -1;
            expectedModCount = coll.modCount;    //Aggiorna il conteggio delle modifiche
        }
    }

        //Implementa il metodo dell'interfaccia Iterable
    public Iterator<E> iterator() {
        return new Itr<E>(this);
    }
}
```

L'implementazione di un iteratore è semplice ma si deve prestare attenzione a due aspetti delicati. Il primo riguarda la possibilità che durante la vita di un iteratore la collezione sia modificata dall'esterno dell'iteratore. Nel nostro esempio, potrebbe essere invocato, durante l'esistenza dell'iteratore, il metodo `add()` o `remove()` (non il metodo omonimo dell'iteratore) oppure il metodo `remove()` di un'altro iteratore. In questi casi non è facile garantire, in generale, che l'iteratore funzioni correttamente, cioè, che visiti tutti gli elementi della collezione una e una sola volta. La soluzione che viene adottata quasi sempre (anche dal framework Collections) consiste nel mantenere un contatore delle modifiche (il campo `modCount`) che permette all'iteratore di "accorgersi" (confrontando `modCount` con il valore del campo `expectedModCount`) se durante la propria esistenza sono avvenute delle modifiche dall'esterno. Se la collezione è stata modificata i metodi dell'iteratore lanciano un'eccezione di tipo `ConcurrentModificationException` (la stessa usata dal framework Collections). Inoltre, l'implementazione del metodo `remove()` dell'iteratore deve aggiornare il conteggio delle modifiche per far sì che un eventuale altro iteratore possa accorgersi della modifica effettuata da questo iteratore. Il secondo aspetto delicato riguarda ancora il metodo `remove()` che deve rimuovere l'elemento ritornato dall'ultima invocazione del metodo `next()`. Per rispettare questo contratto è necessario controllare che ci sia stata una invocazione precedente del metodo `next()` e che nel frattempo non sia già stato invocato il metodo `remove()`. Nel nostro esempio abbiamo semplicemente usato il campo `lastRet` per mantenere l'indice dell'ultimo elemento ritornato dal metodo `next()`, se tale elemento non c'è (perché o `next()` non è stato invocato o `remove()` è già stato invocato) `lastRet` ha valore `-1`.

Prima di passare ad alcuni esempi che illustrano l'uso degli iteratori, conviene vedere una versione avanzata del `for` che Java ha introdotto in concomitanza con l'interfaccia `Iterable` e che risulta molto utile quando si devono scorrere gli elementi di una collezione.

for-each La sintassi di questa forma avanzata di `for`, detto *for-each*, è la seguente:

```
for (Type v : set-expr)
```

```
    <istruzione o blocco di istruzioni>
```

Dove *set-expr* è una qualsiasi espressione il cui valore è o un array o un oggetto che implementa l'interfaccia `Iterable`. Perciò *set-expr* rappresenta la collezione su cui iterare. Il tipo *Type* della variabile `v` deve essere compatibile con quello degli elementi della collezione. Più precisamente, se *set-expr* rappresenta un array allora *Type* deve essere un supertipo del tipo degli elementi dell'array, se invece *set-expr* rappresenta un oggetto che implementa `Iterable<ElemType>` allora *Type* deve essere un supertipo di `ElemType`. L'effetto di questo `for` è che ad ogni iterazione la variabile `v` contiene il prossimo elemento della collezione, e continua così fino a che tutti gli elementi sono stati considerati. Ecco alcuni semplici esempi. Un metodo che ritorna la media dei valori di un array di interi:

```
static double average(int[] values) {
    double sum = 0.0;
    for (int val : values)
        sum += val;
    return sum/values.length;
}
```

Qui il for-each

```
    for (int val : values)
        sum += val;
```

è equivalente al `for`

```
    for (int i = 0 ; i < values.length ; i++)
        sum += values[i];
```

Un metodo generico che ritorna il numero di elementi di un array che si ripetono esattamente `r` volte:

```
static <E> int countRep(E[] a, int r) {
    int count = 0;
    for (E x : a) {    //Per ogni x nell'array a
        int c = 0;
        for (E y : a)    //Conta il numero di ripetizioni di x
            if (x != null ? x.equals(y) : y == null) c++;
        if (c == r) count++;
    }
    return count/r;
}
```

Certo, non sempre il for-each può sostituire il `for`. Se, ad esempio, dobbiamo trovare l'indice in cui si trova un dato valore in un array il for-each non può essere usato. Però nel caso delle collezioni il for-each è quasi sempre più conveniente del `for` tradizionale. Ecco un semplicissimo metodo generico che stampa gli elementi di una collezione:

```
static <E> void print(ACollection<E> coll) {
    for (E x : coll)
        System.out.println(x);
}
```

Qui il for-each è equivalente al `for`

```
    for (Iterator<E> iter = coll.iterator() ; iter.hasNext() ; )
        System.out.println(iter.next());
```

che è decisamente più verboso e meno leggibile del for-each. Il seguente programma contiene vari altri esempi relativi ad una collezione di stringhe:

```
import static java.lang.System.*;

public static void main(String[] args) {
    ACollection<String> frutti = new ACollection<String>();
    frutti.addAll("Mela", "Arancia", "Pera", "Limone", "Melone", "Limone", "Pera");

        //Stampa il numero massimo di ripetizioni di un frutto
    int maxRip = 0;
    for (String f1 : frutti) {
        int c = 0;
        for (String f2 : frutti)
```

```
            if (f1.equals(f2)) c++;
        if (c > maxRip) maxRip++;
    }
    out.println("Numero massimo di ripetizioni: "+maxRip);

        //Rimuove tutti i frutti di lunghezza maggiore di 4
    for (Iterator<String> i = frutti.iterator() ; i.hasNext() ; )
        if (i.next().length() > 4) i.remove();

        //Stampa i frutti rimasti
    for (String f : frutti)
        out.print(f+" ");
    out.println();
}
```

Si osservi che se si vuole usare il metodo `remove()` dell'iteratore non si può usare il for-each. Passiamo ora ad un'altro modo di definire un iteratore.

Implementazione tramite classe locale C'è un modo un po' più semplice di implementare un iteratore. Si può usare una classe interna locale (*local inner class*), detta più brevemente classe locale. Una classe locale può essere definita in un qualsiasi blocco di codice, come il corpo di un metodo, un costruttore, o il blocco di un `for`, `while`, ecc. La classe così definita è locale al blocco esattamente come lo è una variabile dichiarata in un blocco. Quindi una classe locale è inaccessibile dall'esterno del blocco nel quale è stata definita. Però le istanze di tale classe sono oggetti normali che possono essere usati come qualsiasi altro oggetto. Se tali istanze devono essere usate al di fuori del blocco di definizione della classe, allora, probabilmente, la classe estenderà classi o implementerà interfacce visibili dall'esterno. Le principali caratteristiche di una classe locale sono le seguenti.

- Non può avere modificatori di accesso.
- Può accedere a tutti le variabili, campi e metodi che sono visibili nel blocco in cui la classe è definita. L'unica restrizione è che una variabile locale (o un parametro di metodo) sono accessibili solo se sono dichiarati `final`.
- I campi e i metodi di una classe locale nascondono (cioè, rendono inaccessibili) le variabili, i campi e i metodi che hanno lo stesso nome. In particolare, un metodo della classe locale nasconde tutti i metodi con lo stesso nome della classe che la contiene (anche se hanno signature differenti).

Ed ecco una implementazione tramite classe locale dell'iteratore per `ACollection`:

```
import java.util.*;

public class ACollection<E> implements Iterable<E> {
    private E[] array;
    private int size;
    private int modCount = 0;

    //Costruttore e metodi find(), contains(), add(), addAll(), remove() e size() omessi

    private void remove(int i) {
        array[i] = array[--size];
        modCount++;
    }

    public Iterator<E> iterator() {
        final ACollection coll = this;    //Serve per far sì che la classe Itr possa
                                          //accedere al metodo remove(int) di ACollection.
        class Itr implements Iterator<E> {
            private int cursor = 0;
            private int lastRet = -1;
            private int expectedModCount = modCount;
                //Controlla che non siano state fatte delle modifiche alla collezione
            private void checkForModification() {
                if (expectedModCount != modCount)
                    throw new ConcurrentModificationException();
            }
                //Implementa il metodo dell'interfaccia Iterator
            public boolean hasNext() {
                checkForModification();
                return (cursor < size);
```

```
            }
                //Implementa il metodo dell'interfaccia Iterator
            public E next() {
                if (!hasNext())
                    throw new NoSuchElementException();
                lastRet = cursor;
                return array[cursor++];
            }
                //Implementa il metodo dell'interfaccia Iterator
            public void remove() {
                checkForModification();
                if (lastRet == -1)
                    throw new IllegalStateException();
                coll.remove(lastRet);    //Metodo remove(int) della classe ACollection
                cursor = lastRet;
                lastRet = -1;
                expectedModCount = modCount;
            }
        }

        return new Itr();
    }
}
```

L'implementazione è un po' più semplice di quella tramite classe statica perché i campi di `ACollection` sono accessibili direttamente dalla classe locale. L'unica eccezione è il metodo usiliario `remove()` della classe `ACollection` che non è direttamente accessibile perché è nascosto dal metodo omonimo dell'iteratore. Per poter accedere a tale metodo abbiamo introdotto la variabile `coll`.

Quando di una classe locale si crea una sola istanza (ogni volta che si esegue il blocco in cui la classe è definita), come nel caso dell'iteratore, allora c'è un modo ancora più succinto di definirla. In altre parole se l'unico scopo della classe locale è quello di estendere una classe o implementare una interfaccia già esistente creando una sola istanza, allora si può usare una classe interna anonima (*anonymous inner class*) o, più brevemente, classe anonima. Una classe anonima è definita e istanziata simultaneamente in un'unica espressione. La sintassi dell'espressione definitoria-creazionale (che definisce la classe e la istanzia) è leggermente differente a seconda che la classe anonima estenda una classe (astratta) o implementi una interfaccia:

```
    new NomeClasse(<eventuali parametri>) {     //NomeClasse è il nome della classe astratta
        <corpo della classe anonima>            //I parametri sono quelli di un costruttore
    }                                           //della classe astratta

    new NomeInterfaccia() {
        <corpo della classe anonima>
    }
```

Da questa sintassi è chiaro perché sono chiamate classi anonime: non c'è il nome della nuova classe. Inoltre, le classi anonime non possono avere costruttori. Nel caso la classe anonima estenda una classe il costruttore usato è uno di quelli della classe che viene estesa. Nel caso dell'interfaccia il costruttore è quello di default senza parametri. L'espressione definitoria-creazionale può essere usata come una normale espressione creazionale. Ed ecco come può essere implementato il metodo `iterator` usando una classe anonima:

```
    public Iterator<E> iterator() {
        final ACollection coll = this;

        return new Iterator<E>() {
            private int cursor = 0;
            private int lastRet = -1;
            private int expectedModCount = modCount;
                //Controlla che non siano state fatte delle modifiche alla collezione
            private void checkForModification() {
                if (expectedModCount != modCount)
                    throw new ConcurrentModificationException();
            }
                //Implementa il metodo dell'interfaccia Iterator
            public boolean hasNext() {
                checkForModification();
                return (cursor < size);
            }
```

```
            //Implementa il metodo dell'interfaccia Iterator
        public E next() {
            if (!hasNext())
                throw new NoSuchElementException();
            lastRet = cursor;
            return array[cursor++];
        }
            //Implementa il metodo dell'interfaccia Iterator
        public void remove() {
            checkForModification();
            if (lastRet == -1)
                throw new IllegalStateException();
            coll.remove(lastRet);
            cursor = lastRet;
            lastRet = -1;
            expectedModCount = modCount;
        }
    };
}
```

Le classi anonime sono utilissime e usatissime in concomitanza con le librerie di Java per le interfacce utente grafiche (*Graphical User Interface* o GUI).

Esercizi

[Errori_I_1] Il seguente codice Java contiene uno o più errori. Trovare gli errori e spiegarli. In particolare, dire per ogni errore se si verifica in compilazione o durante l'esecuzione.

```
public class Test {
    public static void main(String[] args) {
        ACollection<Integer> coll = new ACollection<Integer>();
        coll.addAll(1, 2, 3, 4, 5, 2, 1);
        for (Integer k : coll) {
            if (k % 2 == 0)
                coll.remove(k);
        }
    }
}
```

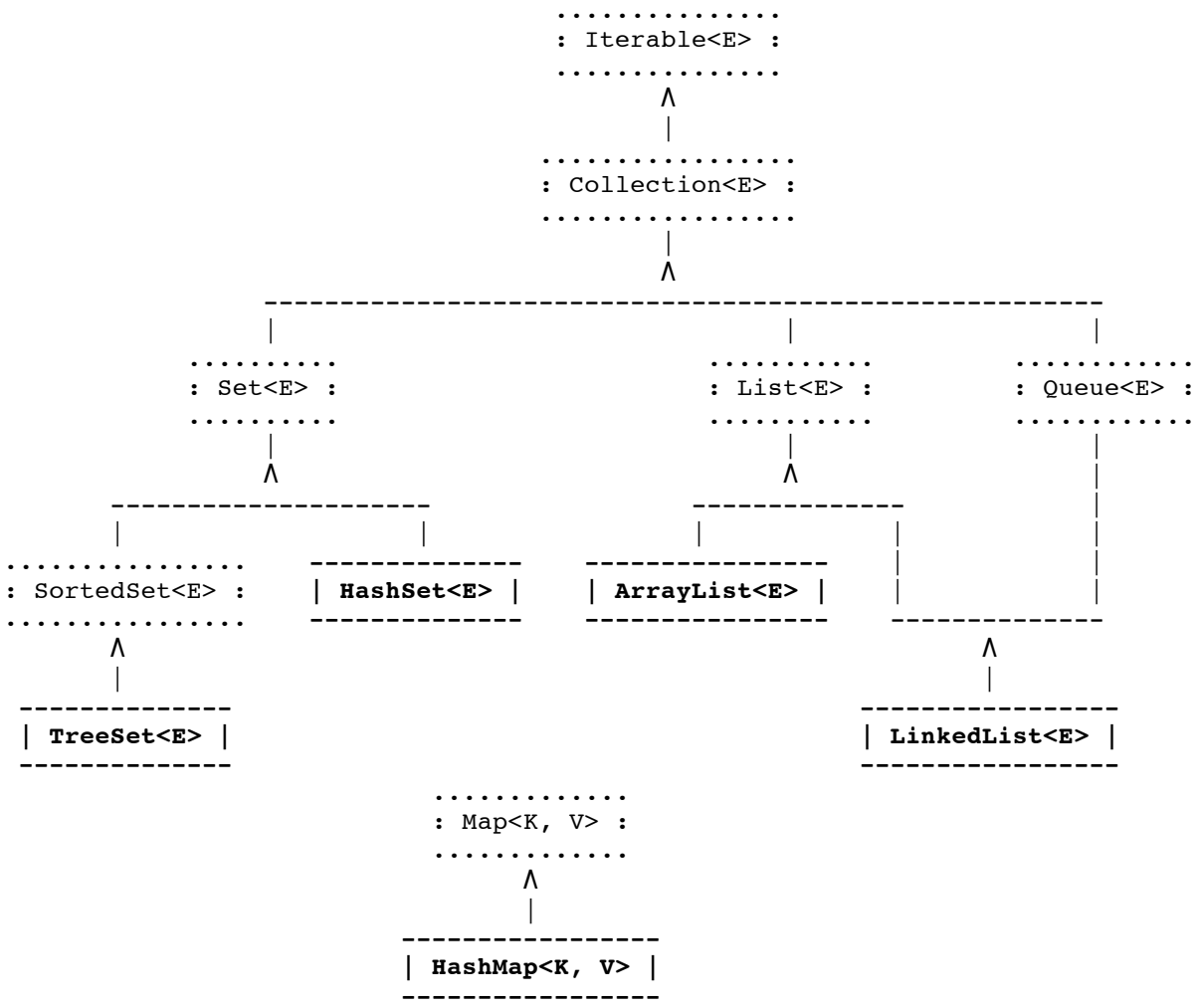
[Errori_I_2] Il seguente codice Java contiene uno o più errori. Trovare gli errori e spiegarli. In particolare, dire per ogni errore se si verifica in compilazione o durante l'esecuzione.

```
public class Test {
    public static void main(String[] args) {
        ACollection<String> coll = new ACollection<String>();
        coll.addAll("A", "AB", "BB", "C", "B");
        for (String s1 : coll) {
            for (Iterator<String> i = coll.iterator() ; i.hasNext() ; ) {
                String s2 = i.next();
                if (!s1.equal(s2) && s1.startsWith(s2))
                    i.remove();
            }
        }
    }
}
```

[Rimuovi_array] Definire un metodo generico che preso in input una collezione di tipo `ACollection` e un array di elementi dello stesso tipo di quelli della collezione, rimuove dalla collezione tutti gli elementi presenti nell'array (comprese tutte le eventuali ripetizioni).
[Collection_comuni] Definire un metodo generico che prese in input due collezioni di tipo `ACollection`, ritorna una nuova collezione, dello stesso tipo, che contiene tutti gli elementi comuni alle due collezioni senza ripetizioni.
[Max_ripetizioni] Definire un metodo generico che presa in input una collezione di tipo `ACollection`, ritorna il numero massimo di ripetizioni di un elemento della collezione.
[Rimuovi_ripetizioni] Definire un metodo che prende in input una collezione di tipo `ACollection` e rimuove da essa tutte le eventuali ripetizioni di elementi.
[Collection_liste] Usare una lista linkata, al posto di un array, per implementare la classe `ACollection`.

Il framework *Collections*

Il framework *Collections*, nel package `java.util`, fornisce un insieme organizzato di classi ed interfacce per gestire collezioni di oggetti di vario tipo. Il seguente diagramma mostra le principali classi ed interfacce del framework (in grassetto sono evidenziate le classi concrete) e le loro relazioni (estensione/implementazione):



Ecco una breve descrizione di ognuna delle interfacce e poi delle classi del diagramma.

`Iterable<E>`
: È l'interfaccia per iterare sugli elementi di una collezione che abbiamo già descritto.

`Collection<E>`
: Interfaccia di base per tutti i tipi di collezione. Definisce i metodi comuni, come `add()`, `contains()`, `remove()`, `size()`, etc. Rappresenta una collezione generica che può contenere anche ripetizioni dello stesso elemento e gli elementi non hanno alcun ordinamento.

`Set<E>`
: Interfaccia per un insieme di elementi (cioè, non può contenere elementi duplicati). Non mantiene alcun ordinamento.

`SortedSet<E>`
: Interfaccia per un insieme ordinato di elementi.

`List<E>`
: Interfaccia per una collezione dove gli elementi sono mantenuti in un ordine determinato dal modo in cui gli elementi sono aggiunti o rimossi.

`Queue<E>`
: Interfaccia per una collezione che mantiene gli elementi ordinati come in una coda (FIFO).

`Map<K,V>`
: Interfaccia per una *mappa*. Una mappa è una collezione di associazioni *(k, v)*, cioè coppie, dove il primo elemento *k* è detto *chiave* (*key*) e il secondo elemento *v* è detto *valore* (*value*). È garantito che non ci sono due associazioni che "mappano" la stessa chiave in due valori (uguali o diversi). Sebbene una mappa possa essere vista come una particolare collezione, non estende l'interfaccia di

base `Collection<E>`.

`HashSet<E>`
: Implementazione dell'interfaccia `Set<E>` tramite tabelle hash.

`ArrayList<E>`
: Implementazione dell'interfaccia `List<E>` tramite array.

`TreeSet<E>`
: Implementazione dell'interfaccia `SortedSet<E>` tramite alberi binari di ricerca bilanciati.

`LinkedList<E>`
: Implementazione delle interfacce `List<E>` e `Queue<E>` tramite liste bidirezionali (*doubly-linked list*). In realtà questa classe implementa una interfaccia che estende `Queue<E>`, chiamata `Deque<E>` (*deque* sta per "*double ended queue*"), la quale permette inserimenti e rimozioni da entrambe le estremità. Quindi permette di rappresentare non solo strutture FIFO, cioè code, ma anche strutture LIFO, cioè pile.

`HashMap<K,V>`
: Implementazione dell'interfaccia `Map<K,V>` tramite tabelle hash.

Consideriamo ora più dettagliatamente tutte queste interfacce e classi.

`Collection<E>` Per prima cosa è bene ricordare che in tutte le collezioni gli elementi sono sempre confrontati tramite il metodo `equals()`. I principali metodi, cioè i più importanti e i più usati, dell'interfaccia di base sono i seguenti.

`boolean` `add(E elem)`
: Aggiunge l'elemento `elem` alla collezione. Se la collezione permette duplicati, il metodo ritorna sempre `true`. Se invece la collezione non permette duplicati (come `Set`), e un elemento equivalente ad `elem` è già presente, il metodo ritorna `false`.

`boolean` `contains(Object elem)`
: Ritorna `true` se la collezione contiene un elemento equivalente a `elem`. Altrimenti ritorna `false`.

`boolean` `remove(Object elem)`
: Rimuove un elemento equivalente a `elem` dalla collezione e ritorna `true` se una occorrenza di tale elemento è stata rimossa. Altrimenti ritorna `false`.

`int` `size()`
: Ritorna il numero di elementi contenuti nella collezione.

`Iterator iterator()`
: Ritorna un iteratore per scandire gli elementi della collezione.

Si sarà notato che i metodi `contains()` e `remove()`, a differenza del metodo `add()`, accettano un `Object` e questo contrasta, in un qualche modo, l'assunzione che la collezione contenga solamente elementi di tipo `E`. Tuttavia, ci sono due ragioni che hanno portato i progettisti del framework ad adottare tale scelta. La prima ha a che fare con l'agevolare la migrazione del codice legacy scritto antecedentemente all'introduzione della genericità. La seconda è il desiderio di rendere l'interfaccia più flessibile in certe circostanze in cui la definizione più stringente dei due metodi (con il tipo `E` al posto del tipo `Object`) può creare delle difficoltà nel suo utilizzo.

`Set<E>` e `HashSet<E>` L'interfaccia `Set<E>` non aggiunge metodi all'interfaccia di base ma semplicemente rende più specifico il contratto di alcuni metodi. I metodi che aggiungono elementi (il metodo `add()` e simili) non permettono di aggiungere duplicati.
La classe `HashSet<E>` implementa l'interfaccia tramite tabelle hash. Il valore hash di un elemento è ottenuto invocando il seguente metodo che è ereditato dalla classe `Object`:

```
public int hashCode()
```

Se il metodo `hashCode()` non è ridefinto ritorna un valore intero che è ricavato dall'indirizzo fisico dell'oggetto e che risulta, generalmente, differente per due oggetti che hanno indirizzi differenti. Perciò, una classe che ridefinisce il metodo `equals()` dovrebbe anche ridefinire il metodo `hashCode()` (se si prevede che gli oggetti della classe possano essere usati in una tabella hash) in modo da garantire che oggetti equivalenti abbiano lo stesso valore hash. Se ci sono elementi equivalenti che non hanno lo stesso valore hash questo può provocare un funzionamento non corretto della tabella hash e quindi anche della classe `HashSet<E>`. Lo stesso discorso vale per qualsiasi implementazione che usa le tabelle hash, come ad esempio `HashMap<K,V>`.
Un semplice esempio che illustra l'uso di `Set<E>` e `HashSet<E>` è il seguente programma che legge dalla linea di comando i pathnames di due files che contengono interi e conta quanti interi del primo file non sono presenti nel secondo file:

```
import java.io.*;
import java.util.*;
import static java.lang.System.*;

public class Example_Set {
        //Ritorna un insieme che contiene gli interi del file pathname
    public static Set<Integer> readSet(String pathname) throws FileNotFoundException {
        Scanner scan = new Scanner(new File(pathname));
        Set<Integer> set = new HashSet<Integer>();
        while (scan.hasNextInt())
            set.add(scan.nextInt());
        return set;
    }
    public static void main(String[] args) throws FileNotFoundException {
        if (args.length != 2) return;
        Set<Integer> set1 = readSet(args[0]);   //Legge il primo insieme
        Set<Integer> set2 = readSet(args[1]);   //Legge il secondo insieme
        int count = 0;
        for (Integer i : set1)                  //Conta gli interi del primo insieme
            if (!set2.contains(i)) count++;     //non contenuti nel secondo.
        if (count == 0)
            out.println("Il primo insieme è contenuto nel secondo");
        else
            out.println("Interi del primo insieme non contenuti nel secondo:"+count);
    }
}
```

`SortedSet<E>` e `TreeSet<E>` L'interfaccia `SortedSet<E>` estende `Set<E>` mantenendo gli elementi dell'insieme ordinati rispetto al loro *ordinamento naturale* (*natural ordering*). L'ordinamento naturale è quello determinato dall'interfaccia `Comparable<E>`. È quindi richiesto che gli elementi inseriti nell'insieme implementino l'interfaccia `Comparable<E>`. L'interfaccia `SortedSet<E>` specializza il contratto dell'iteratore richiedendo che gli elementi siano scanditi secondo il loro ordine naturale. Inoltre, introduce alcuni metodi aggiuntivi di cui i principali sono:

`E first()`
: Ritorna il primo elemento dell'insieme.

`E last()`
: Ritorna l'ultimo elemento dell'insieme.

La classe `TreeSet<E>` implementa l'interfaccia `SortedSet<E>` tramite alberi binari di ricerca bilanciati. Inoltre, aggiunge anche altri metodi perchè implementa una estensione di tale interfaccia che si chiama `NavigableSet<E>`. Un esempio di uso della classe è il seguente programma che legge dalla linea di comando il pathname di un file e due stringhe `s1` e `s2` e poi stampa a video, in modo ordinato, tutte le stringhe (distinte) contenute nelle linee del file che lessicograficamente sono comprese tra `s1` e `s2`.

```
import java.io.*;
import java.util.*;
import static java.lang.System.*;

public class. Example_SortedSet {
    public static void main(String[] args) throws FileNotFoundException {
        if (args.length != 3) return;
        Scanner scan = new Scanner(new File(args[0]));
        SortedSet<String> set = new TreeSet<String>();  //Crea un insieme ordinato
        while (scan.hasNextLine())                      //che contiene le stringhe
            set.add(scan.nextLine());                   //distinte del file.
        out.println("Prima stringa: \""+set.first()+"\"");
        out.println("Ultima stringa: \""+set.last()+"\"");
        String s1 = args[1], s2 = args[2];
        out.println("Stringhe comprese tra \""+s1+"\" e \""+s2+"\":");
        for (String s : set) {              //Stampa le stringhe comprese tra s1 e s2
            if (s.compareTo(s2) > 0) break;
            if (s.compareTo(s1) >= 0)
                out.println(s);
        }
    }
```

```
}
```

`List<E>` e `ArrayList<E>` L'interfaccia `List<E>` estende l'interfaccia di base per rappresentare una collezione in cui ogni elemento è in una ben precisa posizione. Le posizioni sono numerate da `0` a `list.size() - 1`. In altre parole la collezione rappresentata è una sequenza di elementi. Questo richiede che il contratto di alcuni metodi sia raffinato. Ad esempio, il metodo `add()` aggiunge l'elemento alla fine della sequenza. I principali metodi aggiunti dall'interfaccia `List<E>` sono i seguenti:

`void add(int index, E elem)`
: Aggiunge l'elemento `elem` in posizione `index` spostando di una posizione in avanti tutti gli elementi che si trovavano nelle posizioni da `index` in poi.

`E set(int index, E elem)`
: Sostituisce l'elemento in posizione `index` con `elem` e ritorna l'elemento sostituito.

`E get(int index)`
: Ritorna l'elemento in posizione `index`.

`E remove(int index)`
: Rimuove e ritorna l'elemento in posizione `index`. Gli eventuali elementi successivi sono spostati di una posizione indietro.

`Iterator<E> iterator()`
: Ritorna un iteratore che scandisce gli elementi secondo l'ordine della sequenza.

In tutti i casi se l'indice non è compreso tra `0` e `list.size() - 1`, i metodi lanciano una eccezione.
La classe `ArrayList<E>` fornisce una semplice implementazione dell'interfaccia tramite array. L'uso è illustrato dal seguente programma che legge dalla linea di comando i pathnames di due file di testo e controlla se le linee del primo file sono una sottosequenza della sequenza delle linee del secondo file:

```
import java.io.*;
import java.util.*;
import static java.lang.System.*;

public class Example_List {
        //Ritorna una lista che contiene le linee del file pathname
    public static List<String> readList(String pathname) throws FileNotFoundException {
        Scanner scan = new Scanner(new File(pathname));
        List<String> list = new ArrayList<String>();
        while (scan.hasNextLine())          //Aggiunge le linee del file
            list.add(scan.nextLine());      //nella lista, nell'ordine.
        return list;
    }
    public static void main(String[] args) throws FileNotFoundException {
        if (args.length != 2) return;
        List<String> list1 = readList(args[0]);
        List<String> list2 = readList(args[1]);
        int curr = 0, n = list2.size();
        for (String s : list1) {                    //Cerca ogni linea della prima lista
            int i = curr;                           //nella seconda lista a partire dalla
            while (i < n && !s.equals(list2.get(i))) //posizione successiva a quella della
                i++;                                //linea precedente.
            if (i < n) curr = i + 1;
            else {
                out.println("Il primo file NON è una sottosequenza del secondo file");
                return;
            }
        }
        out.println("Il primo file È una sottosequenza del secondo file");
    }
}
```

`Queue<E>` e `LinkedList<E>` L'interfaccia `Queue<E>` estende l'interfaccia di base in modo che la collezione rappresenti una coda, cioè una struttura FIFO. Il contratto del metodo `add()` è raffinato, nel senso che l'elemento è aggiunto in coda. Inoltre, l'interfaccia introduce alcuni metodi:

`boolean offer(E elem)`

Aggiunge l'elemento `elem` in coda e se ha successo ritorna `true`, altrimenti ritorna `false`. Questo metodo è preferibile al metodo `add()` quando la coda può rigettare l'aggiunta di un elemento, ad esempio se è una coda con capacità limitata.

`E peek()`
Ritorna l'elemento di testa della coda (senza rimuoverlo), se la coda è vuota ritorna `null`.

`E poll()`
Ritorna e rimuove l'elemento di testa della coda, se la coda è vuota ritorna `null`.

La classe `LinkedList<E>` implementa l'interfaccia `Queue<E>` tramite una lista linkata. In realtà la classe implementa una estensione dell'interfaccia `Queue<E>` che si chiama `Deque<E>` e rappresenta una struttura più flessibile di una coda in quanto può rappresentare anche una pila. Inoltre `LinkedList<E>` implementa anche l'interfaccia `List<E>`.

`Map<K,V>` e `HashMap<K,V>` L'interfaccia `Map<K,V>` non estende l'interfaccia di base `Collection<E>` perchè ciò che è rappresentato, cioè una mappa, si differenzia da una collezione sotto vari aspetti. La differenza principale sta nel fatto che non si aggiungono elementi ad una mappa ma coppie chiave-valore. Inoltre, una mappa permette di ritrovare il valore associato ad una data chiave. Una chiave o non ha valori associati o ha esattamente un valore associato. Mentre un valore può essere associato a più chiavi. Si pensi, ad esempio, ad una associazione persona-indirizzo. Il nominativo (il codice fiscale) di una persona è una chiave a cui è associato il suo indirizzo di residenza. Chiaramente, ad ogni persona (chiave) possiamo associare al più un solo indirizzo (valore). Mentre, un indirizzo può essere associato a più persone. I principali metodi dell'interfaccia `Map<K,V>` sono i seguenti (`K` è il tipo delle chiavi e `V` è il tipo dei valori):

`V put(K key, V value)`
Aggiunge l'associazione (`key`, `value`) alla mappa. Se la chiave `key` aveva già un valore associato questo è sostituito con il nuovo valore `value` e il vecchio valore è ritornato. Se non c'era una associazione precedente, aggiunge quella nuova e ritorna `null`.

`V get(Object key)`
Ritorna il valore a cui la chiave `key` è associata o `null` se tale valore non c'è.

`boolean containsKey(Object key)`
Ritorna `true` se la mappa contiene una associazione con la chiave `key`. A differenza del metodo `get()` questo metodo permette di distinguere il caso in cui la chiave non ha un valore associato dal caso in cui ha associato il valore `null`.

`V remove(Object key)`
Rimuove l'associazione della chiave `key` e ritorna il valore che era associato alla chiave. Se la chiave non è associata a nessun valore, ritorna `null`.

`Set<K> keySet()`
Ritorna una *vista* (*view*) dell'insieme delle chiavi della mappa. Il fatto che è una vista dell'insieme delle chiavi significa che l'insieme è supportato dalla mappa stessa e cambiamenti della mappa si riflettono sull'insieme e viceversa. Se la mappa viene modificata mentre si sta effettuando una iterazione dell'insieme delle chiavi (eccetto che la modifica sia dovuta al metodo `remove()` dell'iteratore) il risultato dell'iterazione è indefinito. L'insieme ritornato supporta tutte le operazioni eccetto quelle di aggiunta di nuove chiavi.

`int size()`
Ritorna il numero di associazioni contenute nella mappa.

La classe `HashMap<K,V>` implementa l'interfaccia tramite tabelle hash. Quindi valgono le stesse considerazioni che sono state fatte in relazione a `HashSet<E>` per quanto riguarda i metodi `hashCode()` e `equals()`, in questo caso, relativamente agli oggetti chiave.
Vediamo subito un piccolo esempio d'uso delle mappe. Il programma che segue legge dallo standard input un intero `n` e lo fattorizza mantenendo in una mappa i fattori primi con le loro molteplicità che poi stampa a video. Ad esempio, se `n = 4689864` allora il programma stampa `"2^3 3^2 53 1229"`.

```
import java.util.*;
import static java.lang.System.*;

public class Example_Map {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        out.print("Digita un intero: ");
        int n = in.nextInt();
        Map<Integer,Integer> factors = new HashMap<Integer,Integer>();
```

```
        int k = 2;                                          //Iniziando da 2 cerca tutti i
        while (n > 1) {                                     //fattori primi di n e per ognuno
            while (n > 1 && (n % k) == 0) {                 //conta il numero di volte che
                Integer h = factors.get(k);                 //divide n. Mantieni i fattori
                factors.put(k, (h != null ? h + 1 : 1));    //e le loro molteplicità nella
                n /= k;                                     //mappa factors.
            }
            k++;
        }
        for (Integer h : factors.keySet()) {          //Scorri le chiavi della mappa
            Integer m = factors.get(h);               //(cioè i fattori) e stampali
            out.print(h+(m > 1 ? "^"+m : "")+" ");    //insieme con i loro valori
        }                                             //(cioè le molteplicità).
        out.println();
    }
}
```

Le mappe sono utili anche per implementare strutture come alberi o grafi. Il prosssimo esempio mostra proprio una implementazione di un albero (di ordine qualsiasi) i cui nodi possono essere di un tipo parametrico `N`:

```
import java.util.*;

public class HashTree<N> implements Iterable<N> {
    private N root;
    private final Map<N,N> parent = new HashMap<N,N>();
    private final Map<N,HashSet<N>> children = new HashMap<N,HashSet<N>>();
        //Crea un albero con un solo nodo, la radice
    public HashTree(N root) {
        this.root = root;
        parent.put(root, null);
        children.put(root, new HashSet());
    }
        //Aggiunge il figlio child al nodo p
    public boolean addChild(N p, N child) {
        if (p == null || child == null || !parent.containsKey(p) || parent.containsKey(child))
            throw new IllegalArgumentException();
        if (children.get(p).contains(child))
            return false;
        parent.put(child, p);
        children.get(p).add(child);
        children.put(child, new HashSet<N>());
        return true;
    }
        //Ritorna la radice dell'albero
    public N getRoot() {
        return root;
    }
        //Ritorna true se u è un nodo dell'albero
    public boolean isNode(N u) {
        return parent.containsKey(u);
    }
        //Ritorna il genitore del nodo u
    public N getParent(N u) {
        if (!parent.containsKey(u))
            throw new IllegalArgumentException();
        return parent.get(u);
    }
        //Ritorna i figli del nodo u in un insieme creato ex novo
    public Set<N> getChildren(N u) {
        if (!parent.containsKey(u))
            throw new IllegalArgumentException();
        return (Set<N>)children.get(u).clone();
    }
        //Iteratore sui nodi dell'albero
    public Iterator<N> iterator() {
        return parent.keySet().iterator();
    }
}
```

La struttura dell'albero è mantenuta in due mappe. La prima mantiene per ogni nodo l'associazione con il

genitore e la seconda, sempre per ogni nodo, mantiene l'associazione con l'insieme dei figli. Chiaramente, si tratta di un albero senza alcun limite sul numero di figli di un nodo. Ecco un programma che crea un albero di stringhe e lo stampa a video:

```
import java.util.*;
import static java.lang.System.*;

public class Test {
        //Stampa a video l'albero tree
    public static <N> void printTree(HashTree<N> tree) {
        printTreeRecursive(tree, tree.getRoot(), "", false);
    }
        //Metodo ricorsivo che stampa ricorsivamente l'albero tree a partire dal nodo u
    private static <N> void printTreeRecursive(HashTree<N> tree, N u,
                                               String prefix, boolean isLast) {
        if (!prefix.isEmpty()) out.print(prefix + "---");
        out.println(u);
        Set<N> children = tree.getChildren(u);
        if (isLast) {
            prefix = prefix.substring(0, prefix.length() - 1)+ " ";
            if (children.size() == 0) out.println(prefix);
        }
        prefix += "    |";
        for (Iterator<N> i = children.iterator() ; i.hasNext() ; )
            printTreeRecursive(tree, i.next(), prefix, !i.hasNext());
    }
    public static void main(String[] args) {
        HashTree<String> tree = new HashTree<String>("Informatica");
        tree.addChild("Informatica", "Software");
        tree.addChild("Informatica", "Hardware");
        tree.addChild("Hardware", "Memorie");
        tree.addChild("Hardware", "Processori");
        tree.addChild("Hardware", "Architetture");
        tree.addChild("Software", "Sistemi Operativi");
        tree.addChild("Software", "Basi di Dati");
        tree.addChild("Software", "Elaborazione di testi");
        tree.addChild("Software", "Elaborazione di immagini");
        tree.addChild("Software", "Algoritmi");
        tree.addChild("Software", "Linguaggi");
        tree.addChild("Linguaggi", "Imperativi");
        tree.addChild("Linguaggi", "Funzionali");
        tree.addChild("Linguaggi", "Orientati agli Oggetti");
        tree.addChild("Imperativi", "C");
        tree.addChild("Imperativi", "Pascal");
        tree.addChild("Orientati agli Oggetti", "C++");
        tree.addChild("Orientati agli Oggetti", "Java");
        tree.addChild("Orientati agli Oggetti", "Smalltalk");
        tree.addChild("Basi di Dati", "SQL");
        tree.addChild("Basi di Dati", "Data Mining");
        tree.addChild("Sistemi Operativi", "Unix");
        tree.addChild("Sistemi Operativi", "Linux");
        tree.addChild("Sistemi Operativi", "MacOS X");
        printTree(tree);
    }
}
```

Il risultato dell'esecuzione del programma è il seguente:

```
Informatica
    |---Software
    |    |---Basi di Dati
    |    |    |---SQL
    |    |    |---Data Mining
    |    |
    |    |---Elaborazione di immagini
    |    |---Sistemi Operativi
    |    |    |---Linux
    |    |    |---Unix
    |    |    |---MacOS X
    |    |
    |    |---Linguaggi
    |    |    |---Imperativi
```

```
|     |     |     |---Pascal
|     |     |     |---C
|     |     |
|     |     |---Funzionali
|     |     |---Orientati agli Oggetti
|     |           |---C++
|     |           |---Java
|     |           |---Smalltalk
|     |
|     |---Algoritmi
|     |---Elaborazione di testi
|
|---Hardware
      |---Processori
      |---Architetture
      |---Memorie
```

Il framework Collections contiene molte altre interfacce e classi. Qui abbiamo descritto solamente quelle fondamentali e di uso più comune.

Esercizi

[Conta_ripetizioni] Scrivere un metodo generico `<E> Map<E,Integer> countRep(Collection<? extends E> coll)` che ritorna una mappa che associa ad ogni elemento distinto della collezione `coll` il numero di volte che occorre nella collezione.
[Parole_frequenti] Scrivere un programma che legge dalla linea di comando il pathname di un file di testo e un intero `n` e stampa a video le `n` parole più frequenti del file.
[ListToMap] Definire un metodo generico che prende in input una lista generica (di tipo `List`) e crea e ritorna una mappa le cui chiavi sono gli elementi distinti della lista e il valore associato ad una chiave `k` è la lista degli indici della lista di input in cui l'elemento `k` compare. Il tipo ritornato non deve dipendere dall'implementazione della mappa o delle liste. Ecco un esempio:

```
Lista di input (lista di stringhe)     Mappa di output
("A", "B", "A", "C", "C", "D")         {("A", (0, 2)), ("B", (1)), ("C", (3, 4)), ("D", (5))}
```

[Lista_di_liste] Definire un metodo generico che prende in input una matrice e crea e ritorna una lista di liste (di tipo `List`) che rappresenta la matrice. Ogni sottolista contiene gli elementi della corrispondente riga della matrice di input. La matrice può avere righe di lunghezze differenti. Il tipo ritornato non deve dipendere dalla particolare implementazione usata per le liste.
[Threshold] Definire un metodo statico generico che prende in input una mappa generica `m` (di tipo `Map<...>`) e un valore `t` dello stesso tipo dei valori della mappa e ritorna una nuova mappa dello stesso tipo di `m` che contiene esattamente tutte le associazioni di `m` che hanno valore maggiore od uguale alla soglia `t`. Ecco un esempio:

```
Mappa di input Map<String, Integer> e soglia 5        Mappa di output
{("A", 2), ("B", 5), ("C", 4), ("D", 6), ("F", 6)}    {("B", 5), ("D", 6), ("F", 6)}
```

Ovviamente, il metodo deve richiedere che il tipo dei valori implementi in modo opportuno l'interfaccia `Comparable`.
[Anagrammi_in_file] Scrivere un programma che legge dalla linea di comando il pathname di un file di testo e stampa a video i gruppi di almeno due parole del file che sono tra loro anagrammi. Ad esempio, se il file è il seguente:

```
La trota apre la bocca sulla pera mentre la torta era mangiata dal ratto.
Eppur le rape erano toste e senza nesso, e questo testo era privo di senso.
```

Allora, il programma stampa:

```
[ratto, trota, torta]
[pera, apre, rape]
[toste, testo]
[nesso, senso]
```

Suggerimento: usare una mappa in cui le chiavi sono le parole con i caratteri ordinati e il valore di una chiave è l'insieme delle

parole che sono anagrammi della chiave.

[Grafi] Definire una classe generica `DGraph<N>` per rappresentare grafi diretti i cui nodi sono di tipo parametrico `N`. Definire metodi per aggiungere nodi, archi, per scandire gli adiacenti (uscenti) di un nodo, per rimuovere archi e nodi.

Suggerimento: usare una mappa per rappresentare il grafo tramite liste di adiacenza: le chiavi sono i nodi e il valore associato ad un nodo è l'insieme dei suoi adiacenti (uscenti).

[Cammino_minimo] Scrivere un metodo generico `<N> List<N> shortestPath(DGraph<N> g, N u, N v)` che ritorna in una lista, creata ex novo, i nodi di un cammino minimo (cioè, con il numero minimo di nodi) nel grafo `g` che va dal nodo `u` al nodo `v`. Il tipo `DGraph<N>` è quello definito nell'esercizio precedente.

Suggerimento: effettuare una visita in ampiezza del grafo a partire dal nodo `u`. Usare una `LinkedList` per mantenere, durante la visita, la coda dei nodi visitati i cui adiacenti devono ancora essere visitati.