Dal linguaggio C al linguaggio Java

(Terza parte)

Riccardo Silvestri

30-3-2009 10-3-2010

Sommario della terza parte

Estendere e condividere

A volte accade che una classe implementi delle funzionalità che vorremmo poter riusare in un'altra classe. Ma non vorremmo dover implementare di nuovo queste funzionalità nella nuova classe. Si pensi, ad esempio, a una classe `Persona` che permette di gestire tramite opportuni metodi le generalità di una persona (nome, cognome, data di nascita), l'indirizzo e magari altro ancora. Dovendo definire una o più classi per gestire i dati relativi ai dipendenti di una azienda sarebbe conveniente poter riusare la classe `Persona`. Un dipendente è anche una persona e i dati gestiti dalla classe `Persona` devono essere gestiti dall'archivio. Si potrebbe definire una classe `Dipendente` che relativamente ai dati in comune con quelli gestiti dalla classe `Persona` contiene una copia della corrispondente implementazione e interfaccia (campi e metodi). Ma se non abbiamo il codice sorgente della classe `Persona`? E anche se avessimo il codice sorgente, nella maggior parte delle situazioni, non è conveniente replicare il codice.

Sempre continuando con il nostro esempio dell'archivio dei dipendenti, sicuramente avremo bisogno di gestire i dati di dipendenti che rivestono ruoli diversi. Ad esempio, potrebbero esserci dei dipendenti nel ruolo di dirigenti. In relazione ad un dirigente si dovranno gestire delle informazioni ulteriori, ad esempio, la denominazione del reparto diretto, eventuali responsabilità di progetto, ecc. Allora diventa naturale definire una nuova classe chiamata appunto `Dirigente`. Però un dirigente è anche un dipendente e così tutti i dati gestiti dalla classe `Dipendente` dovranno essere gestiti anche dalla classe `Dirigente`. Cosa facciamo? Replichiamo il codice della classe `Dipendente` nella classe `Dirigente`? Chiaramente questa non è la soluzione ottimale.

Per fortuna Java, al pari di quasi tutti i lunguaggi orientati agli oggetti, fornisce un meccanismo che

permette di definire una nuova classe estendendo una classe esistente. La nuova classe, così definita, eredita tutti i campi e i metodi (accessibili) della classe originale senza bisogno di replicarne il codice. Quindi la classe `Dirigente` può essere definita come una estensione della classe `Dipendente` (e la classe `Dipendente` può, a sua volta, estendere la classe `Persona`). La classe `Dirigente` necessiterà solamente dell'implementazione dei campi e metodi che servono per gestire i dati che sono di esclusiva pertinenza di un dirigente ma non di un dipendente generico. Così la classe `Dirigente` eredita, e quindi *condivide*, l'implementazione e l'interfaccia della classe `Dipendente`. È anche vero che la classe `Dirigente` *estende* la classe `Dipendente` perchè definisce metodi e campi che la classe `Dipendente` non possiede. Inoltre, il tipo della classe che estende un'altra classe diventa un sottotipo di quest'ultima. Così il tipo della classe `Dirigente` diventa un sottotipo della classe `Dipendente`. Questo significa che ovunque si può usare un oggetto di tipo `Dipendente` si può anche usare un oggetto di tipo `Dirigente`.

Una classe che ne estende un'altra non solo eredita l'interfaccia e l'implementazione ma può anche modificare il comportamento dei metodi che eredita. Se, ad esempio, la classe `Dipendente` ha un metodo `salario()` che ritorna appunto il salario del dipendente, la classe `Dirigente` può ridefinire il metodo in modo che ritorni il salario del dirigente, mantenendo la stessa interfaccia. Questo, insieme al fatto che il tipo della classe `Dirigente` è trattato come un sottotipo della classe `Dipendente`, è un esempio di ciò che viene chiamato *polimorfismo* (sarà spiegato tra poco).

Ereditarietà e polimorfismo

La sintassi di Java per definire una classe che ne estende un'altra è molto semplice e consiste nell'uso della parola chiave `extends` nell'intestazione della classe. Consideriamo come primo esempio una classe `Title` che serve a rappresentare dei titoli, cioè, delle stringhe che possono venir stampate in una varietà di modi. Per rendere l'esempio semplice ci limiteremo a definire la classe così che permetta di stampare la stringa variando solamente la spaziatura tra i caratteri.

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

public class Title {
    private String title;
    private int spacing;

    public Title(String t) {
        title = t;
        spacing = 0;          // per default lo spazio tra i caratteri è zero
    }

    public void setSpacing(int space) { spacing = space; }

    public int printLength() {    // ritorna il numero di caratteri stampati
        return title.length()*(1 + spacing) - spacing;
    }

    public void print() {         // stampa il titolo
        int nc = title.length();
        for (int i = 0 ; i < nc ; i++) {
            out.print(title.charAt(i));
            if (i < nc - 1)
                for (int j = 0 ; j < spacing ; j++) out.print(' ');
        }
    }
}
```

Consideriamo ora una classe `AlignTitle` che estende la classe `Title` aggiungendo la possibilità di rappresentare titoli con allineamento a sinistra, a destra o centrale rispetto ad un campo di lunghezza specificata. In questo e nei prossimi esempi supporremo che la definizione della classe sia sempre preceduta dalla direttiva `import static java.lang.System.*;`.

```
public class AlignTitle extends Title {
    public static final int LEFT = 0, CENTER = 1, RIGHT = 2;

    private int alignment, fSize;
```

```
    public AlignTitle(String t, int align, int field) {
        super(t);            // invoca il costruttore della classe Title
        alignment = align;
        fSize = field;
    }

    public void setAlign(int align) { alignment = align; }
    public void setField(int size) { fSize = size; }

    public void print() {    // ridefinisce (override) il metodo della classe Title
        int dLen = printLength();    // invoca un metodo ereditato
        int left = 0, right;
        switch (alignment) {
            case LEFT: left = 0; break;
            case CENTER: left = (fSize - dLen)/2; break;
            case RIGHT: left = fSize - dLen; break;
        }
        right = fSize - printLength() - left;
        for (int i = 0 ; i < leftChars ; i++) out.print(' ');
        super.print();         // invoca il metodo print() della classe Title
        for (int i = 0 ; i < right ; i++) out.print(' ');
    }
}
```

Superclasse/sottoclasse Con l'intestazione `class AlignTitle extends Title` si dichiara che la classe `AlignTitle` estende la classe `Title`. Questo significa che tutti i membri accessibili (campi e metodi) della classe `Title` sono ereditati dalla classe `AlignTitle`. I membri privati e i costruttori non sono invece ereditati e i membri privati non sono neanche accessibili. Ad esempio, i metodi `setSpacing()` e `print()` fanno automaticamente parte dei metodi pubblici di `AlignTitle`, con l'implementazione definita nella classe `Title`. In questo modo la classe `AlignTitle` eredita sia l'interfaccia della classe `Title` sia l'implementazione. Nella terminologia comune la classe che viene estesa è detta *classe base* o *superclasse* e la classe che estende è detta *classe derivata* o *sottoclasse*. Così `Title` è la classe base (o superclasse) della classe `AlignTitle` e la classe `AlignTitle` è una classe derivata da (o una sottoclasse di) `Title`.

Costruttori e la parola chiave `super` Siccome i costruttori non sono ereditati, una sottoclasse deve necessariamente definire almeno un costruttore. L'unica eccezione a questa regola si ha quando la superclasse ha un costruttore senza argomenti: in questo caso per la sottoclasse è implicitamente definito un costruttore di default senza argomenti che automaticamente invoca il costruttore senza argomenti della superclasse. Se però la superclasse, come nel caso di `Title`, non ha un costruttore senza argomenti, allora la sottoclasse deve definire almeno un costruttore. Inoltre, ogni costruttore della sottoclasse deve invocare un costruttore della superclasse. Ciò è necessario perché un oggetto istanza della sottoclasse è anche un oggetto istanza della superclasse e così deve essere costruito in relazione ad entrambe le classi. Per effettuare la costruzione dell'oggetto rispetto alla superclasse si usa la parola chiave `super` che può essere interpretata come il riferimento alla parte dell'oggetto che è di esclusiva pertinenza della superclasse. L'invocazione del costruttore della superclasse deve sempre essere la prima istruzione. Nel nostro esempio il costruttore di `AlignTitle` deve invocare `super(t)` per costruire la parte dell'oggetto che riguarda la superclasse `Title` (cioè, l'inizializzazione dei campi `title` e `spacing`).

Ridefinire (override) Ovviamente, la sottoclasse può definire nuovi membri. Nel nostro esempio i campi `alignment` e `fSize` e i metodi `setAlign()` e `setField()`. Inoltre può *ridefinire* (*override*) i metodi che eredita. La classe `AlignTitle` ridefinisce il metodo `print()` della sua superclasse `Title` perchè deve gestire l'allineamento del titolo. Però può riusare l'implementazione originale della superclasse all'interno della nuova implementazione. La parola chiave `super` permette di accedere al metodo della superclasse. Così `super.print()` invoca proprio il metodo `print()` della superclasse `Title`. Si osservi che in questa invocazione è necessario usare la parola chiave `super` perché altrimenti si sarebbe invocato il metodo stesso, cioè il metodo `print()`, definendo così un metodo ricorsivo che in esecuzione produce un ciclo infinito. Mentre l'invocazione del metodo `printLength()` non necessita dell'uso della parola chiave `super` perchè non vi è ambiguità: `printLength()` non è stato ridefinito.

Polimorfismo Come si è detto, una sottoclasse eredita l'interfaccia della superclasse. Inoltre, il tipo

definito dalla sottoclasse è considerato un *sottotipo* (*subtype*) del tipo della superclasse. Ciò significa che ovunque si può usare un oggetto della superclasse si può anche usare un oggetto della sottoclasse. Nel nostro esempio, il tipo `TitleAlign` è un sottotipo di `Title` e ovunque si può usare un oggetto di tipo `Title` si può anche usare un oggetto di tipo `AlignTitle`. Ad esempio, se un metodo richiede come parametro un oggetto di tipo `Title` allora gli si può passare anche un oggetto di tipo `AlignTitle` (il viceversa non è possibile perchè, ad esempio, il metodo `setAlign()` appartiene all'interfaccia del sottotipo `AlignTitle` ma non appartiene all'interfaccia del supertipo `Title`). Questa caratteristica è detta *polimorfismo*. Un oggetto di tipo `AlignTitle` può assumere diverse "forme" potendo essere usato sia come un oggetto `AlignTitle` sia come un oggetto `Title`. Il seguente semplice programma usa le classi `Title` e `AlignTitle` e mostra il polimorfismo degli oggetti di tipo `AlignTitle`.

```
public class Test {
    public static void stampaTitoli(Title[] array) {
        for (int i = 0 ; i < array.length ; i++) {
            array[i].print();
            out.println();
        }
        out.println();
    }
    public static void main(String[] args) {
        Title t = new Title("Classe Base");
        AlignTitle ta = new AlignTitle("Ereditarietà", AlignTitle.RIGHT, 50);
        Title[] array = new Title[3];
        array[0] = t;
        array[1] = ta;
        array[2] = new AlignTitle("Polimorfismo", AlignTitle.CENTER, 50);
        stampaTitoli(array);
        for (int i = 0 ; i < array.length ; i++)
            array[i].setSpacing(1);
        stampaTitoli(array);
        ta.setAlign(AlignTitle.CENTER);
        ((AlignTitle)array[2]).setAlign(AlignTitle.RIGHT);
        stampaTitoli(array);
    }
}
```

Si noti come è possibile creare un oggetto di tipo `AlignTitle` e assegnarlo ad una variabile di tipo `Title` (senza un cast esplicito). Il metodo `stampaTitoli()` stampa un array di oggetti di tipo `Title` però alcuni di questi sono di tipo `AlignTitle`. Quando il metodo `print()` è invocato, l'implementazione che deve essere usata (quella della classe `Title` o quella della classe `AlignTitle`) è determinata in base all'identità (il riferimento) dell'oggetto su cui è invocato il metodo. L'identità di un oggetto contiene infatti anche il nome della classe a cui l'oggetto appartiene. La tecnica che permette di determinare l'implementazione di un metodo durante l'esecuzione è detta *dynamic binding* (cioè, legame dinamico). Questa è in contrapposizione allo *static binding* (legame statico) che invece determina l'implementazione staticamente, al momento della compilazione. Chiaramente, il polimorfismo richiede necessariamente l'uso del dynamic binding. Si osservi che quando un oggetto del sottotipo `AlignTitle` è mantenuto in una variabile del supertipo `Title` si può usare un cast per invocare su di esso metodi che appartengono alla sottoclasse ma non alla superclasse (ad esempio il metodo `setAlign()`).

Già da questo semplicissimo esempio si può notare come il polimorfismo aiuti a trattare in modo uniforme oggetti di tipo diverso. Il metodo `stampaTitoli()` tratta senza distinzioni oggetti di tipo `Title` e di tipo `AlignTitle` senza però sacrificarne le differenze. Infatti la stampa degli oggetti, grazie al polimorfismo, usa automaticamente per ogni oggetto l'implementazione appropriata. Ed ecco, infine, il risultato dell'esecuzione del programma:

```
Classe Base
                                    Ereditarietà
                  Polimorfismo

C l a s s e   B a s e
                         E r e d i t a r i e t à
              P o l i m o r f i s m o

C l a s s e   B a s e
              E r e d i t a r i e t à
                         P o l i m o r f i s m o
```

Sottotipi e array In questo primo esempio abbiamo visto una classe (`AlignTitle`) che ne estende un'altra (`Title`). Dovrebbe essere chiaro che la classe che estende può a sua volta essere estesa da una ulteriore classe. Invero, non c'è nessun limite sul numero di classi che possono esserci in una catena in cui ogni classe estende quella che la precede. Consideriamo, ad esempio, la seguente catena con tre classi:

```
class A { ... }
class B extends A { ... }
class C extends B { ... }
```

Quindi la classe `C` estende la classe `B` che a sua volta estende la classe `A`. La terminologia delle superclassi/sottoclassi è usata anche in relazione a classi che non sono direttamente l'una l'estensione dell'altra. Così si dice che `C` è una sottoclasse di `A` (oltre a essere una sottoclasse di `B`) e che `A` è una superclasse di `C` (oltre a essere anche una superclasse di `B`). L'importante concetto di sottotipo si applica parimenti a tutte le sottoclassi dirette o indirette di una classe. Quindi `C` è, al pari di `B`, un sottotipo di `A`. Ovunque si può usare un oggetto di tipo `A` si può anche usare un oggetto di tipo `C`.

La relazione di sottotipo si estende anche agli array. Se *`Type`* è un qualsiasi tipo e *`Subtype`* è un sottotipo di *`Type`* allora *`Subtype`*`[]` è un sottotipo di *`Type`*`[]`. Così `B[]` e `C[]` sono sottotipi di `A[]` e `C[]` è un sottotipo di `B[]`. Ecco alcuni esempi:

```
A[] arrayA;
B[] arrayB;
C[] arrayC = new C[10];
arrayA = arrayC;       // OK perché C[] è un sottotipo di A[]
arrayB = arrayC;       // OK perché C[] è un sottotipo di B[]
arrayB = arrayA;       // ERRORE (in compilazione) A[] non è un sottotipo di B[]
arrayA = (B[])arrayC;  // OK C[] è un sottotipo di B[] che è un sottotipo di A[]
```

La relazione di sottotipo si estende naturalmente anche agli array di array:

```
A[][] matrixA;
B[][] matrixB;
C[][] matrixC = new C[5][10];
matrixA = matrixC;     // OK C[][] è un sottotipo di A[][]
matrixB = matrixC;     // OK C[][] è un sottotipo di B[][]
matrixB = matrixA;     // ERRORE (in compilazione) A[][] non è un sottotipo di B[][]
matrixA = (B[][])matrixC;  // OK C[][] è un sottotipo di B[][] che è un sottotipo di A[][]
```

Si noti che il fatto che `C[][]` è un sottotipo di `A[][]` deriva da: `C` è un sottotipo di `A` e questo implica che `C[]` è un sottotipo di `A[]` che a sua volta implica che `C[][]` è un sottotipo di `A[][]`.

Essenzialmente la relazione di sottotipo (tra tipi classe o tipi array) corrisponde alle conversioni cast che sono corrette in compilazione (cioè, il compilatore non segnala alcun errore). Più precisamente se *`Type1`* e *`Type2`* sono due tipi classe o array e `var` è una variabile di tipo *`Type2`*, allora la conversione cast `(`*`Type1`*`)var` è corretta in compilazione se e solo se *`Type2`* è un sottotipo di *`Type1`*.

Esercizi

[Errori_tipi] Il seguente programma contiene 4 errori (uno di questi si verifica solamente in esecuzione), trovarli e spiegarli.

```
class Point {
    public final double x, y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
class LPoint extends Point {
    public final String label;

    public LPoint(double x, double y, String l) {
```

```
        label = l;
        super(x, y);
    }
}
public class Test {
    public static void main(String[] args) {
        Point[] pA = new Point[10];
        pA[0] = new LPoint(0, 0, "Roma");
        System.out.println(pA[0].label);
        LPoint[] lpA = new Point[5];
        Point[][] pM = new LPoint[5][];
        pM[0] = new Point[5];
    }
}
```

[Titoli_incorniciati] Definre una classe `FrameTitle` che estende la classe `AlignTitle` e permette di stampare (tramite il metodo `print()`) titoli con cornice come nei seguenti esempi:

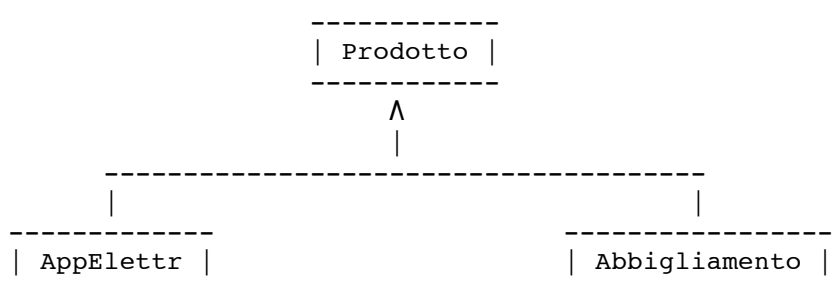
```
**************      **********************      **********************
*Polimorfismo*      *Polimorfismo        *      *    Polimorfismo    *
**************      **********************      **********************
```

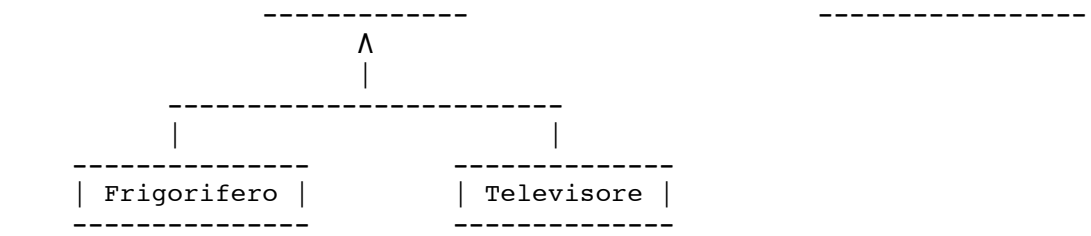
Potrebbe essere conveniente modificare la classe `AlignTitle` ridefinendo opportunamente il metodo `printLength()`. Si potrebbe definire un metodo che permette di modificare il carattere della cornice `'*'`.
[Titoli_verticali] Ripensare le classi `Title` e `AlignTitle` in modo tale che possano stampare anche titoli in verticale.
Suggerimento: Aggiungere alla classe `Title` le costanti `HORIZONTAL` e `VERTICAL`, un campo `direction`, i metodi `setDirection()` e `getDirection()` e modificare opportunamente il metodo `print()` così che stampi il titolo in orizzontale o verticale a seconda del valore del campo `direction`. Poi la classe `AlignTitle` si occuperà dell'allineamento per titoli orizzontali o verticali. Per i titoli verticali le costanti `LEFT`, `CENTER` e `RIGHT` possono essere interpretate, rispettivamente, come *top*, *center* e *bottom*.

Gerarchie di classi

Consideriamo ora un esempio un po' più elaborato. Si immagini una situazione in cui si deve gestire un archivio di prodotti merceologici di varia natura come elettrodomestici, televisori, capi di abbigliamento, ecc. Ogni oggetto dell'archivio dovrebbe rappresentare una specifica tipologia di prodotto. Ad esempio, un televisore di una certa marca e modello o una camicia di un certa marca, taglia e colore. Chiaramente ci sono degli attributi (o proprietà) che sono comuni a tutti i prodotti: prezzo e produttore. Altri attributi non sono comuni a tutti i prodotti ma appartengono a una certa categoria di prodotti. Ad esempio, il consumo in watt è comune a tutti i prodotti elettrici (elettrodomestici, televisori, ecc.).

Possiamo organizzare proprio in base a queste comunanze e differenze le classi per la gestione di questo archivio. Definiamo una classe base `Prodotto` per la gestione degli attributi comuni a tutti i prodotti. Poi definiamo delle estensioni di tale classe per le diverse categorie specifiche di prodotti. Per mantenere l'esempio semplice consideriamo solamente poche categorie: abbigliamento, frigoriferi e televisori. I capi di abbigliamento possono essere gestiti da una sola classe che chiamiamo appunto `Abbigliamento`. Quindi `Abbigliamento` sarà una sottoclasse di `Prodotto`. I frigoriferi e i televisori hanno alcuni attributi in comune (ad esempio, consumo in watt) però hanno anche delle differenze: la capacità ha senso solamente per i frigoriferi e la dimensione in pollici ha senso solamente per i televisori. Così conviene definire una classe intermedia, che chiamiamo `AppElettr`, che accomuna tutti i prodotti elettrici o elettronici. È anch'essa una sottoclasse di `Prodotto`. Infine definiamo le classi `Frigorifero` e `Televisore` come sottoclassi di `AppElettr`. Possiamo descrivere visivamente le relazioni tra queste classi tramite il seguente diagramma:





Come si vede le relazioni di estensione tra classi producono una gerarchia di classi. Le classi che si trovano più in alto sono quelle più generali e via via che si scende si trovano classi sempre più specializzate. La relazione di estensione può quindi anche essere vista come una relazione di specializzazione. Ad esempio, la classe `Frigorifero` specializza `AppElettr` che a sua volta specializza `Prodotto`. Ovviamente, le relazioni possono anche essere viste nel verso opposto e quindi, ad esempio, la classe `Prodotto` generalizza `AppElettr` che a sua volta generalizza sia `Frigorifero` che `Televisore`. Iniziamo col definire la classe base `Prodotto`.

```
public class Prodotto {
    private float prezzo;
    private String produttore;

    public Prodotto(float p, String prod) {
        prezzo = p;
        produttore = prod;
    }

    public String getProduttore() { return produttore; }

    public float prezzoAlConsumo() { return prezzo; }

    public void stampa() {
        out.println("  Produttore: "+produttore);
        out.println("  Prezzo: "+prezzo+" euro");
    }
}
```

Il metodo `stampa()` produce una stampa degli attributi del prodotto. Questo comportamento deve essere rispettato da tutte le classi nella gerarchia che essendo più specializzate hanno attributi aggiuntivi e dovranno quindi necessariamente ridefinire il metodo. Passiamo ora alla definizione della classe `Abbigliamento` che deriva direttamente da `Prodotto`.

```
public class Abbigliamento extends Prodotto {
    private String categoria;    // il tipo del capo (camicia, pantaloni, ecc.)
    private int taglia;
    private String colore = null;

    public Abbigliamento(float p, String prod, String cat, int t) {
        super(p, prod);    // costruttore della classe Prodotto
        categoria = cat;
        taglia = t;
    }

    public void setColore(String c) { colore = c; }

        // ridefinisce il metodo della classe Prodotto
    public void stampa() {
        out.println(categoria);
        super.stampa();    // invoca il metodo della classe Prodotto
        out.println("  Taglia: "+taglia);
        if (colore != null) out.println("  Colore: "+colore);
    }
}
```

Si noti come il costruttore estenda il costruttore della classe `Prodotto` aggiungendo altri attributi e come il costruttore della superclasse è invocato. Il metodo `stampa()` è ridefinito per stampare anche gli attributi propri dei capi di abbigliamento. Il metodo della superclasse è però comunque invocato (`super.stampa()`) per la stampa degli attributi comuni. Passiamo ora alla definizione della classe

`AppElettr` che deriva anch'essa direttamente da `Prodotto`.

```
public class AppElettr extends Prodotto {
    private float contributoRAEE;    // per il riciclo degli app. elettrici e elettronici
    private String modello;
    private int consumoWatt = 0;

    public AppElettr(float p, String prod, float raee, String mod) {
        super(p, prod);    // costruttore della classe Prodotto
        contributoRAEE = raee;
        modello = mod;
    }

        // ridefinisce il metodo della classe Prodotto
    public float prezzoAlConsumo() {
        return super.prezzoAlConsumo() + contributoRAEE;
    }

    public String getModello() { return modello; }
    public void setConsumoWatt(int watt) { consumoWatt = watt; }

        // ridefinisce il metodo della classe Prodotto
    public void stampa() {
        out.println("  Produttore: "+getProduttore());
        out.println("  Prezzo (compreso contributo RAEE): "+prezzoAlConsumo()+" euro");
        if (consumoWatt > 0) out.println("  Consumo: "+consumoWatt+" watt");
    }
}
```

La classe `AppElettr` ridefinisce il metodo `prezzoAlConsumo()` perchè deve aggiungere il contributo RAEE (Riciclo Apparecchi Elettrici e Elettronici) che è comune a tutti i prodotti di questa classe (l'importo però varia a seconda della tipologia). Si noti come anche in questo caso si usa la parola chiave `super` per accedere al metodo della superclasse. Il metodo `stampa()` è ridefinito senza poter sfruttare il metodo della classe `Prodotto` perché la dicitura dell'attributo prezzo è differente. Inoltre, si noti che l'attributo `modello` non è stampato perché questo è stampato in modo particolare dalle sottoclassi di `AppElettr` (sempre per questa ragione c'è il metodo `getModello()`). Infine definiamo le classi `Frigorifero` e `Televisore` che derivano direttamente da `AppElettr`.

```
public class Frigorifero extends AppElettr {
    private int capacita;    // capacità in litri del frigorifero

    public Frigorifero(float p, String prod, float raee, String mod, int cap) {
        super(p, prod, raee, mod);    // costruttore della classe AppElettr
        capacita = cap;
    }
        // ridefinisce il metodo della classe AppElettr
    public void stampa() {
        out.println("FRIGORIFERO: "+getModello());
        super.stampa();    // invoca il metodo della classe AppElettr
        out.println("  Capacità: "+capacita+" litri");
    }
}


public class Televisore extends AppElettr {
    private int pollici;

    public Televisore(float p, String prod, float raee, String mod, int pol) {
        super(p, prod, raee, mod);    // costruttore della classe AppElettr
        pollici = pol;
    }
        // ridefinisce il metodo della classe AppElettr
    public void stampa() {
        out.println("TELEVISORE: "+getModello());
        super.stampa();    // invoca il metodo della classe AppElettr
        out.println("  Dimensione: "+pollici+" pollici");
    }
}
```

Si noti come in entrambe le classi il metodo `stampa()` è stato ridefinito e come l'attributo `modello` è stato stampato. Consideriamo anche un programmino che mette alla prova le classi appena definite.

```
public class Test {
    public static void stampaProdotti(Prodotto[] array, float maxPrezzo) {
        out.println("PRODOTTI CON PREZZO NON SUPERIORE A "+maxPrezzo+" EURO");
        for (int i = 0 ; i < array.length ; i++) {
            if (array[i].prezzoAlConsumo() <= maxPrezzo) {
                array[i].stampa();
                out.println();
            }
        }
        out.println();
    }
    public static void testProdotto() {
        Prodotto[] array = new Prodotto[4];
        array[0] = new Frigorifero(1000, "AEGO", 10, "FF234Q", 320);
        array[1] = new Televisore(800, "LGS", 6.5f, "TVP001SD", 40);
        array[2] = new Abbigliamento(45, "Abba", "CAMICIA", 42);
        array[3] = new Abbigliamento(70, "Levit", "PANTALONI", 48);
        stampaProdotti(array, 1000);
        ((AppElettr)array[0]).setConsumoWatt(2000);
        ((AppElettr)array[1]).setConsumoWatt(150);
        ((Abbigliamento)array[2]).setColore("Celeste");
        ((Abbigliamento)array[3]).setColore("Marrone");
        stampaProdotti(array, 1200);
    }
}
```

Grazie al polimorfismo i diversi prodotti una volta creati possono poi essere trattati in modo uniforme perchè ereditano tutti l'interfaccia definita nella classe base `Prodotto`. Ed ecco il risultato dell'esecuzione:

```
PRODOTTI CON PREZZO NON SUPERIORE A 1000.0 EURO
TELEVISORE: TVP001SD
  Produttore: LGS
  Prezzo (compreso contributo RAEE): 806.5 euro
  Dimensione: 40 pollici

CAMICIA
  Produttore: Abba
  Prezzo: 45.0 euro
  Taglia: 42

PANTALONI
  Produttore: Levit
  Prezzo: 70.0 euro
  Taglia: 48


PRODOTTI CON PREZZO NON SUPERIORE A 1200.0 EURO
FRIGORIFERO: FF234Q
  Produttore: AEGO
  Prezzo (compreso contributo RAEE): 1010.0 euro
  Consumo: 2000 watt
  Capacita': 320 litri

TELEVISORE: TVP001SD
  Produttore: LGS
  Prezzo (compreso contributo RAEE): 806.5 euro
  Consumo: 150 watt
  Dimensione: 40 pollici

CAMICIA
  Produttore: Abba
  Prezzo: 45.0 euro
  Taglia: 42
  Colore: Celeste
```

```
PANTALONI
  Produttore: Levit
  Prezzo: 70.0 euro
  Taglia: 48
  Colore: Marrone
```

Che cosa abbiamo imparato da questi primi esempi? Una cosa che, sopratutto da quest'ultimo esempio, si può intuire è che l'ereditarietà, se ben usata, permette di strutturare il codice in modi che rispecchiano la natura delle informazioni reali che si vogliono rappresentare. La gerarchia delle classi rispecchia in modo fedele la gerarchia di concetti che sorge naturalmente nel momento in cui si vogliono organizzare le informazioni relative ai diversi prodotti. La gerarchia permette anche di raccogliere facilmente a fattore comune tutto ciò che si può condividere. Così che le classi si differenziano solamente dove è veramente necessario. Questo elimina alla radice le possibili duplicazioni di codice rendendo più facile il controllo della correttezza e la manutenzione. Inoltre, la struttura gerarchica, raccogliendo a fattor comune tutto ciò che è condivisibile, facilita l'estensione delle funzionalità del sistema. Ad esempio, se si vuole aggiungere una classe per rappresentare le lavatrici, basterà introdurre una sottoclasse di `AppElettr` e gestire in essa solo ciò che differenzia le lavatrici dagli altri apparecchi elettrici/elettronici (quello che è in comune come il prezzo, il produttore, il consumo in watt, ecc. è già gestito dalle superclassi).

Un'altro importante vantaggio offerto dalla ereditarietà e in particolar modo dal polimorfismo sta nella possibilità di trattare in modo uniforme oggetti di natura diversa. Il metodo `stampaProdotti()` tratta in modo uniforme oggetti che rappresentano frigoriferi, televisori e capi di abbigliamento, mantenendo al contempo la specificità di ogni oggetto. Questo, in ultima analisi, aiuta a scrivere codice più snello, più leggibile e meno soggetto ad errori.

Esercizi

[Errori_prodotti] Il seguente programma usa le definizioni della gerarchia precedentemente definita e contiene 3 errori (uno solo dei quali si verifica in compilazione), trovarli e spiegarli.

```
public class Test {
    public static void main(String[] args) {
        Prodotto[][] pM = new AppElettr[10][];
        pM[0] = new Televisore[5];
        pM[1] = new Frigorifero[8];
        pM[2] = new Abbigliamento[10];
        System.out.println(pM[0][0].getProduttore());
        pM[1][0] = new Frigorifero(1000, "AAA", 8, "FF", 300);
        System.out.println(pM[1][0].getModello());
        ((Frigorifero)pM[1][0]).setConsumoWatt(500);
    }
}
```

[Estensione_prodotti] Definire una classe `Lavatrice` estendendo la classe `AppElettr` per gestire attributi specifici come *capacità di carico* (in Kg) e *tipo caricamento* (frontale o superiore). Ovviamente la stampa effettuata dal metodo `stampa()` deve essere coerente con quella degli altri apparecchi elettrici.

[Estensione_prodotti+] Definire una piccola gerarchia di classi che estende la classe `AppElettr` per gestire i dati relativi a computer. Si consideri la possibilità di definire una classe `Computer` che raccoglie gli attributi comuni a tutti i tipi di computer e poi delle sottoclassi per *desktop*, *portatili* (notebook), ed eventualmente anche per *ultraportatili*.

[File_system] Definire una piccola gerarchia di classi per gestire le informazioni relative ai file e alle directory di un file system.

Suggerimento: Definire una classe base `FS_Item` per rappresentare le informazioni comuni ai file e alle directory (nome, path assoluto, diritti di accesso, ecc.) e poi una sottoclasse per i *file* e una sottoclasse per le *directory*. Prevedere anche un metodo `isDir()` della classe base che ritorna `true` solo se l'oggetto è una directory.

[Regioni] Si vuole realizzare un archivio per mantenere i dati relativi alle *regioni*, *provincie* e *capoluoghi di provincia*. Per ogni regione si vuole gestire il nome, l'estensione (in Km quadrati), la popolazione e i collegamenti alle provincie. Per ogni provincia si vuole gestire il nome, l'estensione, la popolazione, il numero di comuni e il collegamento al capoluogo di provincia. Per ogni capoluogo di provincia si vuole gestire il nome, la popolazione, l'estensione e l'elenco dei nomi di tutte le circoscrizioni. Definire una gerarchia di classi per la rappresentazione dell'archivio secondo le seguenti

indicazioni. Definire una classe base `ElemGeo` che gestisce i dati comuni ai diversi elementi (regioni, provincie e capoluoghi) e un codice numerico che identifica univocamente ogni elemento. Definire poi le sottoclassi `Regione`, `Provincia` e `Capoluogo` per gestire i dati specifici. Definire opportuni metodi per impostare i dati ed eventualmente leggerli. Definire un metodo `stampa` che stampa tutti i dati di un elemento.

[Figure_geometriche] Definire una classe `Punto` per rappresentare punti del piano a coordinate intere. Definire una gerarchia di classi per gestire figure geometriche del piano (cerchi, rettangoli e triangoli) a coordinate intere secondo le indicazioni seguenti.

a. Definire una classe base `Figura2D` e le sottoclassi `Cerchio`, `Rettangolo` e `Triangolo`. Ogni oggetto di tipo `Cerchio` è determinato da un centro di tipo `Punto` e un raggio (intero). Ogni oggetto `Rettangolo` è determinato da due punti (di tipo `Punto`) rappresentanti lo spigolo in alto a sinistra e quello in basso a destra (il rettangolo ha i lati paralleli agli assi). Ogni oggetto `Triangolo` è determinato da tre punti (i vertici del triangolo). Definire un metodo `area` che ritorna l'area della figura geometrica. Definire un metodo `minR` che ritorna un oggetto `Rettangolo` che è il più piccolo rettangolo che contiene la figura geometrica. Definire inoltre un metodo `isIn` che prende in input un punto (di tipo `Punto`) e ritorna `true` o `false` a seconda che il punto cada all'interno o all'esterno della figura geometrica.
b. Definire un metodo statico `maxArea` della classe `Figura2D` che prende in input un array di `Figura2D` e ritorna la massima area delle figure geometriche dell'array.
c. Definire un metodo statico `minRettangolo` della classe `Figura2D` che prende in input un array di `Figura2D` e ritorna un oggetto `Rettangolo` che è il più piccolo rettangolo che contiene tutte le figure geometriche dell'array.

[Biblioteca] Si vuole gestire un archivio dei documenti (libri e DVD) di una biblioteca. Ogni documento ha una collocazione. Prima di tutto definire quindi una classe `Collocazione` per gestire appunto le collocazioni. Una collocazione è determinata da una stringa che specifica il nome di un reparto della biblioteca, un numero di scaffale che identifica uno scaffale del reparto e da un numero che indica una posizione nello scaffale. Poi, definire una gerarchia di classi secondo le seguenti indicazioni.

a. Definire una classe base `Documento` e poi le sottoclassi `Libro`, `DVD_Video` e `DVD_Audio`. Un oggetto di tipo `Libro` consiste in una collocazione (un oggetto di tipo `Collocazione`), una stringa contenente l'autore o gli autori del libro, una stringa contenente il titolo e un intero contenente il numero di pagine. Un oggetto `DVD_Video` consiste in una collocazione, una stringa che contiene il titolo del film, una stringa che contiene il regista o i registi e un intero che contiene la durata in minuti del film. Un oggetto `DVD_Audio` consiste in una collocazione, una stringa che contiene il nome della casa discografica, una stringa che contiene il titolo del DVD e per ogni brano una stringa contenente il titolo del brano. Definire un metodo `stampa` che stampa le informazioni relative ad un documento. Definire un metodo `cercaInTitolo` che prende in input una stringa `str` e ritorna `true` o `false` a seconda che `str` sia contenuta o meno nel titolo del documento.
b. Definire un metodo statico `cercaTitoli` della classe `Documento` che prende in input un array di oggetti `Documento arrD` e una stringa `str` e ritorna in un array di oggetti `Documento` tutti i documenti dell'array `arrD` il cui titolo contiene la stringa `str`.

La classe `Object`

Nel linguaggio Java tutte le classi estendono automaticamente, direttamente o indirettamente, una classe speciale chiamata **`Object`**. Quindi tutte le classi sono sottoclassi dirette o indirette di questa classe. Quando una qualsiasi classe è definita, anche se non estende esplicitamente nessuna classe, implicitamente estende la classe `Object`. Se ad esempio definiamo una classe `NomeClasse`:

```
class NomeClasse {
    ...
}
```

Ciò è equivalente a scrivere:

```
class NomeClasse extends Object {
```

```
    ...
}
```

Non solo tutte le classi sono quindi dei sottotipi del tipo `Object` ma anche qualsiasi tipo array è un sottotipo del tipo `Object`. Questo ha due importanti effetti il primo è che una variabile di tipo `Object` può mantenere il riferimento ad un qualsiasi oggetto sia esso una istanza di una classe o di un array. Il secondo effetto è che tutti gli oggetti ereditano i metodi della classe `Object`. Dapprima discuteremo il primo effetto e poi il secondo.

Consideriamo alcuni esempi che mostrano come il tipo `Object` possa essere visto come l'equivalente del tipo `void *` (cioè, un puntatore generico) del linguaggio C:

```
Title title = new Title("Polimorfismo");
Object obj = title;     // OK perché Title è un sottotipo di Object
obj = "stringa";        // OK perché String è un sottotipo di Object
obj = null;             // OK
int[] interi = new int[10];
obj = interi;           // OK perché l'array int[] è un sottotipo di Object
obj = new Title[5];     // OK perché l'array Title[] è un sottotipo di Object
title = obj;            // ERRORE (in compilazione) Title non è un supertipo di Object
interi = obj;           // ERRORE (in compilazione) int[] non è un supertipo di Object
```

Si noti che il tipo `Object[]` (array di `Object`) è un supertipo di tutti i tipi *`Type`*`[]` tali che *`Type`* è un sottotipo di `Object`. Quindi `Object[]` è il supertipo di tutti i tipi array eccetto gli array di tipi primitivi, dato che `Object` non è un supertipo di nessuno dei tipi primitivi. Vediamo alcuni esempi:

```
Object[] objArray;
objArray = new Title[10];       // OK
String[][] matrixStr = new String[10[20];
objArray = matrixStr;           // OK
int[] interi = new int[10];
objArray = interi;              // ERRORE in compilazione
int[][] matrixInt = new int[20][30];
objArray = matrixInt;           // OK
```

La ragione per cui `object[]` è un supertipo di `String[][]` è che `Object` è un supertipo di `String[]`. Come si vede dall'errore segnalato per l'assegnamento `objArray = interi`, `Object[]` non è un supertipo di `int[]` perché `Object` non è un supertipo di `int`. Però `Object[]` è un supertipo di `int[][]` dato che `Object` è un supertipo di `int[]`. E questo spiega la ragione per cui l'assegnamento `objArray = matrixInt` è corretto.

Nonostante, come abbiamo detto, `Object` non è un supertipo di nessuno dei tipi primitivi i seguenti assegnamenti sono corretti:

```
Object obj = 15;      // OK
float v = 0.34;
obj = v;              // OK
obj = true;           // OK
char c = 'A';
obj = c;              // OK
```

Come è possibile ciò? La ragione sta nel fatto che il compilatore Java, in tutti i contesti come questi, esegue una conversione automatica che converte il tipo primitivo in un corrispondente oggetto (ad esempio un `int` è convertito in un oggetto di tipo `Integer`). Questa conversione, che è chiamata auto-boxing, sarà trattata più avanti.

La classe `Object` introduce un modo generico e uniforme per riferirsi ad oggetti dei tipi più vari. Questo è utile per scrivere metodi che possono operare in modo uniforme su oggetti di tipo diverso. Supponiamo di voler scrivere un metodo che copia un array su un'altro array. Una possibile implementazione è la seguente:

```
public static void copiaArray(Object[] src, Object[] dst) {
    int n = (src.length <= dst.length ? src.length : dst.length);
    for (int i = 0 ; i < n ; i++)
        dst[i] = src[i];
}
```

Quindi il metodo `copiaArray(src, dst)` copia i valori delle componenti dell'array `src` nelle componenti dell'array `dst` facendo attenzione alle lunghezze dei due array. Il metodo può essere usato

per copiare array di `String`, array di `Title` array di `Point`, ecc. Come nei seguenti esempi:

```
String[] a = {"primo", "secondo", "terzo"};
String[] b = new String[5];
copiaArray(a, b);
Title[] t = {new Title("Classe"), new Title("Oggetto")};
Title[] tt = new Title[10];
copiaArray(t, tt);
AlignTitle[] at = {new AlignTitle("A", AlignTitle.LEFT, 8),
                   new AlignTitle("B", AlignTitle.LEFT, 8)};
copiaArray(at, t);    // OK perché il tipo di at è un sottotipo del tipo di t
int[][] mat = {{1, 2, 3}, {4, 5, 6}};
int[][] mat2 = new int[4][5];
copiaArray(mat, mat2);
```

Ovviamente non può essere usato per copiare array di tipi primitivi. Se il tipo effettivo (cioè, il tipo al run-time) di `src` non è un sottotipo del tipo effettivo di `dst` allora accade un errore al run-time che produce una eccezione di tipo `ArrayStoreException`. Ad esempio

```
copiaArray(a, mat);
```

non produce alcun errore in compilazione ma produrrà un errore in esecuzione.

L'operatore `instanceof` Se si vuole definire un metodo che permette di fare anche la copia di array di tipi primitivi è necessario scrivere un metodo con la seguente intestazione:

```
public static void copiaArray(Object src, Object dst)
```

Infatti questo metodo può accettare come argomenti array di `int` o di un qualsiasi tipo primitivo. Però per poterlo implementare è necessario avere la possibilità di riconoscere il tipo effettivo degli argomenti. Per questo Java mette a disposizione l'operatore **`instanceof`**. Per un qualsiasi tipo riferimento (classe o array) *`Type`* e un qualsiasi riferimento ad un oggetto (istanza di una classe o di un array) *`ref`*, l'espressione

`ref` **`instanceof`** *`Type`*

è `true` se e solo se il tipo al run-time di *`ref`* è un sottotipo di (o è uguale a) *`Type`*. In altre parole, l'espressione ha valore `true` se e solo se l'oggetto *`ref`* o è una istanza del tipo *`Type`* o è una istanza di qualche sottotipo di *`Type`*. Vediamo subito alcuni esempi chiarificatori:

```
Object obj = new Object();
if (obj instanceof Object) {...}       // VERO
if (obj instanceof Object[]) {...}     // FALSO
String str = "A";
if (str instanceof String) {...}       // VERO
if (str instanceof Object) {...}       // VERO
if (str instanceof Object[]) {...}     // ERRORE in compilazione
if (obj instanceof String) {...}       // FALSO
obj = str;
if (obj instanceof String) {...}       // VERO
Title titolo = new Title("A");
if (titolo instanceof AlignTitle) {...}       // FALSO
titolo = new AlignTitle("B", AlignTitle.LEFT, 8);
if (titolo instanceof AlignTitle) {...}       // VERO
if (titolo instanceof Title) {...}            // VERO
int[] interi = new int[10];
if (interi instanceof Object[]) {...}         // ERRORE in compilazione
obj = interi;
if (obj instanceof Object[]) {...}            // FALSO
if (obj instanceof int[]) {...}               // VERO
if (obj instanceof long[]) {...}              // FALSO
if (interi instanceof long[]) {...}           // ERRORE in compilazione
```

Si osservi che l'espressione `str instanceof Object[]` produce immediatamente un errore in compilazione perchè il compilatore può determinare che ha sempre valore `false`. Invece l'espressione `obj instanceof String` può avere, a seconda del valore al run-time della variabile `obj`, sia valore `true` che `false`.

Vediamo ora come l'operatore `instanceof` può essere usato per implementare il metodo `copiaArray()`:

```
public static void copiaArray(Object src, Object dst) {
    if ((src instanceof Object[]) && (dst instanceof Object[])) {
        Object[] s = (Object[])src;
        Object[] d = (Object[])dst;
        int n = (s.length <= d.length ? s.length : d.length);
        for (int i = 0 ; i < n ; i++) d[i] = s[i];
    } else if ((src instanceof boolean[]) && (dst instanceof boolean[])) {
        boolean[] s = (boolean[])src;
        boolean[] d = (boolean[])dst;
        int n = (s.length <= d.length ? s.length : d.length);
        for (int i = 0 ; i < n ; i++) d[i] = s[i];
    } else if ((src instanceof byte[]) && (dst instanceof byte[])) {
        byte[] s = (byte[])src;
        byte[] d = (byte[])dst;
        int n = (s.length <= d.length ? s.length : d.length);
        for (int i = 0 ; i < n ; i++) d[i] = s[i];
    } else if ((src instanceof short[]) && (dst instanceof short[])) {
        short[] s = (short[])src;
        short[] d = (short[])dst;
        int n = (s.length <= d.length ? s.length : d.length);
        for (int i = 0 ; i < n ; i++) d[i] = s[i];
    } else if ((src instanceof int[]) && (dst instanceof int[])) {
        int[] s = (int[])src;
        int[] d = (int[])dst;
        int n = (s.length <= d.length ? s.length : d.length);
        for (int i = 0 ; i < n ; i++) d[i] = s[i];
    } else if ((src instanceof long[]) && (dst instanceof long[])) {
        long[] s = (long[])src;
        long[] d = (long[])dst;
        int n = (s.length <= d.length ? s.length : d.length);
        for (int i = 0 ; i < n ; i++) d[i] = s[i];
    } else if ((src instanceof char[]) && (dst instanceof char[])) {
        char[] s = (char[])src;
        char[] d = (char[])dst;
        int n = (s.length <= d.length ? s.length : d.length);
        for (int i = 0 ; i < n ; i++) d[i] = s[i];
    } else if ((src instanceof float[]) && (dst instanceof float[])) {
        float[] s = (float[])src;
        float[] d = (float[])dst;
        int n = (s.length <= d.length ? s.length : d.length);
        for (int i = 0 ; i < n ; i++) d[i] = s[i];
    } else if ((src instanceof double[]) && (dst instanceof double[])) {
        double[] s = (double[])src;
        double[] d = (double[])dst;
        int n = (s.length <= d.length ? s.length : d.length);
        for (int i = 0 ; i < n ; i++) d[i] = s[i];
    } else {
        throw new IllegalArgumentException("Array di tipo diverso");
    }
}
```

Ed ecco alcuni esempi di invocazione di tale metodo:

```
int[] intA = {0,1,2,3,4};
int[] intB = new int[10];
copiaArray(intA, intB);
double[] dA = {2.897, 0.0067, 2.3459};
double[] dB = {0, 2, 6.7, 5.78986};
copiaArray(dA, dB);
String[] strA = {"cane", "gatto", "topo"};
String[] strB = new String[6];
copiaArray(strA, strB);
AlignTitle[] atA = {new AlignTitle("A", AlignTitle.LEFT, 8),
                    new AlignTitle("B", AlignTitle.LEFT, 8)};
Title[] tB = new Title[4];
copiaArray(atA, tB);
copiaArray(intA, dB);   // ERRORE in esecuzione: IllegalArgumentException
```

```
copiaArray(strA, tB);   // ERRORE in esecuzione: ArrayStoreException
```

Si noti che `copiaArray(atA, tB)` è corretta nonostante il tipo di `atA` (`AlignTitle[]`) è diverso dal tipo di `tB` (`Title[]`) perché `AlignTitle[]` è un sottotipo di `Title[]`. La piattaforma Java fornisce dei metodi di utilità per gli array. Ad esempio la classe `System` ha il seguente metodo statico:

```
public static void arraycopy(Object src, int srcPos, Object dest,
                                         int destPos, int length)
```

È simile al nostro `copiaArray()` ma permette in più di specificare gli indici di inizio e fine sia del sottoarray da copiare che di quello in cui copiare. Nel package `java.util` c'è la classe `Arrays` che ha un gran numero di metodi di utilità per gli array.

Dopo aver discusso le potenzialità offerte dal fatto che una variabile di tipo `Object` può mantenere il riferimento ad un qualsiasi oggetto (classe o array), è giunto il momento di discutere gli effetti relativi al fatto che tutti gli oggetti ereditano i metodi della classe `Object`. La classe `Object` ha parecchi metodi alcuni dei quali riguardano i threads che non tratteremo, mentre alcuni altri non siamo ancora pronti ad affrontarli. Così limiteremo la discussione solamente a due metodi: `equals()` e `toString()`. Questi sono anche quelli di uso più comune.

Il metodo `equals()` L'intestazione del metodo `equals()` è la seguente:

```
public boolean equals(Object obj)
```

L'implementazione di default (cioè quella fornita direttamente dalla classe `Object`) non è di grande utilità perché semplicemente ritorna `true` se e solo se il riferimento di `obj` è uguale a quello dell'oggetto su cui il metodo è invocato. Ecco un semplice esempio:

```
class IntPoint {
    public int x, y;
    public IntPoint(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
public class Test {
    public static main(String[] args) {
        IntPoint p1 = new IntPoint(1, 1);
        intPoint p2 = new IntPoint(1, 1);
        if (p1.equals(p2)) {         // FALSO perchè p1 e p2 sono oggetti diversi
            ...                      // anche se hanno lo stesso valore
        }
        if (p1.equals(p1)) {...}     // sempre VERO
    }
}
```

È chiaro quindi che se si vuole che gli oggetti di una classe implementino una versione significativa del metodo `equals()`, cioè una versione che controlla l'uguaglianza di valore (non di identità), è necessario che il metodo sia ridefinito. infatti tutte le classi della piattaforma Java per le quali ha senso usare il metodo `equals()` lo ridefiniscono. Ad esempio, la classe `String`. Consideriamo la ridefinizione del metodo per la classe `IntPoint`:

```
class IntPoint {
    public int x, y;
    public IntPoint(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public boolean equals(Object obj) {  // ridefinisce il metodo equals() di Object
        if (obj == null) return false;
        if (!(obj instanceof IntPoint))  // controlla che abbia il tipo IntPoint
            return false;
        IntPoint p = (IntPoint)obj;
        return (x == p.x && y == p.y);   // controlla che abbia lo stesso valore
    }
}
```

Se adesso rifacciamo un test con la nuova classe `IntPoint` otteniamo:

```
public class Test {
    public static main(String[] args) {
        IntPoint p1 = new IntPoint(1, 1);
        intPoint p2 = new IntPoint(1, 1);
        if (p1.equals(p2)) {          // VERO perchè sono oggetti dello stesso tipo
            ...                       // IntPoint e hanno lo stesso valore (anche se
        }                             // sono oggetti diversi)
        p2.x = 2;
        if (p1.equals(p2)) {...}      // FALSO oggetti dello stesso tipo IntPoint
    }                                 // ma con valori differenti
}
```

La ridefinizione del metodo `equals()` che abbiamo dato non controlla che l'oggetto `obj` sia un'instanza della classe `IntPoint` ma solamente che sia un'istanza di una qualche sottoclasse (tramite l'operatore `instanceof`). Avremmo dovuto invece controllare che era proprio della stessa classe? Questo è un punto piuttosto delicato e non c'è una risposta univoca. In alcuni casi è più conveniente definirlo come sopra e in altri conviene controllare l'uguaglianza della classe (e c'è un modo di farlo). Non approfondiremo oltre l'argomento perché al momento sarebbe prematuro.

Il metodo `toString()` L'intestazione del metodo `toString()` è la seguente:

```
public String toString()
```

Il metodo ritorna una rappresentazione tramite stringa dell'oggetto su cui è invocato. Come al solito l'implementazione di default non è molto utile. Infatti, ritorna una stringa contenente il nome della classe dell'oggetto seguita dal carattere `'@'` e poi la rappresentazione in esadecimale di un codice hash dell'oggetto (per ora non approfondiremo da dove proviene e a cosa serve questo codice). Ecco alcuni esempi:

```
public class Test {
    public static main(String[] args) {
        IntPoint p1 = new IntPoint(1, 1);
        intPoint p2 = new IntPoint(1, 1);
        out.println(p1.toString());
        out.println(p2.toString());
        p2.x = 2;
        out.println(p2.toString());
        Title t = new Title("Titolo");
        out.println(t.toString());
    }
}
```

Il risultato dell'esecuzione è il seguente (assumendo che le classi `IntPoint` e `Title` siano nel package `metodologie`):

```
metodologie.IntPoint@dbe178
metodologie.IntPoint@af9e22
metodologie.IntPoint@af9e22
metodologie.Title@b6ece5
```

Quindi, come nel caso del metodo `equals()`, se si vuole che tale metodo sia utile è necessario ridefinirlo. Tutte le classi della piattaforma Java per cui il metodo `toString()` è utile lo ridefiniscono. Ad esempio la classe `String` (ritorna la stringa stessa). Il metodo `toString()` è importante anche perché è automaticamente invocato (dal compilatore) tutte le volte che il riferimento ad un oggetto è usato in una espressione di concatenazione di stringhe come operando dell'operatore `+`. Ad esempio l'espressione `"punto: "+p1` è automaticamente trasformata dal compilatore nell'espressione `"punto: "+p1.toString()`. Inoltre il metodo `println()` quando riceve come argomento il riferimento ad oggetto invoca il metodo `toString()` su quell'oggetto. Infatti nel programma precedente avremmo potuto scrivere `out.println(p1)` invece di `out.println(p1.toString())`. Vediamo ora come si può ridefinire il metodo `toString()`. Per semplicità consideriamo le classi `IntPoint` e `Title`:

```
class IntPoint {
    ...                     // la parte che rimane invariata è omessa
    public String toString() {
```

```
        return "("+x+", "+y+")";
    }
}
class Title {
    ...                     // la parte che rimane invariata è omessa
    public String toString() {
        return title;
    }
}
```

Se ora eseguiamo di nuovo il programma di test precedente otteniamo il seguente risultato:

```
(1, 1)
(1, 1)
(2, 1)
Titolo
```

Tutti i metodi della classe `Object` sono anche ereditati dagli oggetti di tipo array. Però, a differenza degli oggetti di tipo classe, per gli oggetti array i metodi non possono essere ridefiniti. Per questa ragione la classe `Arrays` ha metodi statici che sono dei validi sostituti per gli array di gran parte dei metodi della classe `Object`, come `toString()` e `equals()`.

Esercizi

[Errori_O_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 Pair {
    private String key, value;
    public Pair(String k, String v) {
        key = k;
        value = v;
    }
    public String getKey() { return key; }
}
public class Test {
    public static void main(String[] args) {
        Pair[] pp = new Pair[] {new Pair("K", "V"), new Pair("KK", "VV")};
        System.out.println(pp[0].toString());
        System.out.println(pp.toString());
        Object[] oA = pp;
        String k = oA[0].getKey();
        Object[] oB = new int[4];
    }
}
```

[Errori_O_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 void main(String[] args) {
        String[] sA = new String[] {"A", "B", "C"};
        double[] dA = new double[] {0.9, 1.2};
        System.out.println(sA.toString()+dA.toString());
        Object[] oA = dA;
        Object obj = sA;
        Object obj2 = dA;
        boolean[][] tab = new boolean[4][4];
        Object[] oB = tab;
        Object[][] oT = tab;
    }
}
```

[Errori_O_3] 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 void main(String[] args) {
        if (args instanceof String) return;
        float[][] matrix = new float[5][];
        Object[] oA = matrix;
        if (oA instanceof float[]) return;
        oA = new int[10];
        Object[] oB = new int[5][4];
        oA = matrix;
        oA[0] = oB[0];
    }
}
```

[Titoli_Object] Ridefinire in modo appropriato i metodi `equals()` e `toString()` per le classi `Title` e `AlignTitle`. È possibile sfruttare le implementazioni dei due metodi per la classe `Title` per le implementazioni relative alla sottoclasse `AlignTitle`?

[Prodotti_Object] Ridefinire in modo appropriato i metodi `equals()` e `toString()` per tutte le classi della gerarchia `Prodotti`. Si possono sfruttare le implementazioni delle superclassi per le implementazioni relative alle sottoclassi?

[Stampa_array] Definire un metodo `stampaArray(Object arr)` che stampa l'array `arr`, se `arr` è effettivamente il riferimento ad un oggetto array, altrimenti non fa nulla (oppure lancia un eccezione appropriata). Il metodo deve trattare sia array di tipi primitivi che array di riferimenti. In quest'ultimo caso deve semplicemente invocare il metodo `toString()` su ogni componente dell'array. Ecco alcuni esempi:

```
ARRAY                                stampaArray(a)
int[] a = {0, 1, 2, 3};              [0, 1, 2, 3]
String[] a = {"il", "la", "lo"};     [il, la, lo]
Title a = {new Title("A"),
           new Title("B")};          [metodologie.Title@cf8583, metodologie.Title@4693c7]
```

[Stampa_multiarray] Definire un metodo `stampaMultiArray(Object arr)` uguale a quello dell'esercizio precedente che però nel caso `arr` è un array di array riapplica la procedura di stampa in modo ricorsivo fino a quando non arriva a componenti che non sono di tipo array (che vengono stampate così come lo sono dal metodo `stampaArray(Object arr)`). Ecco alcuni esempi:

```
ARRAY                                                stampaMultiArray(a)
int[][] a = {{0,1,2},{3,4},{5,6,7,8}};               [[0, 1, 2], [3, 4], [5, 6, 7, 8]]
String[][] a = {{"il","lo","la"},{"un","una"}};      [[il, lo, la], [un, una]]
byte[][][] a = {{{1,2},{3,4}},{{5},{6,7,8}}};        [[[1, 2], [3, 4]], [[5], [6, 7, 8]]]
```

Classi astratte

Accade piuttosto spesso che una classe base di una gerarchia di classi non possa fornire l'implementazione di alcuni metodi perché non esiste alcuna implementazione significativa al livello della classe base. Però tali metodi devono comunque essere definiti nella classe base perchè questo permette di trattare poi gli oggetti delle varie sottoclassi in modo uniforme sfruttando il polimorfismo. Si pensi a una gerarchia di classi per il disegno (su finestre grafiche) di varie figure geometriche (rettangoli, rettangoli con spigoli tondi, cerchi, ellissi, ecc.). Probabilmente sarebbe una buona scelta di progettazione prevedere una classe base `Shape` che rappresenta la radice della gerarchia. Le sottoclassi rappresenteranno le specifiche figure geometriche (una classe per i rettangoli, una per i rettangoli con spigoli tondi, ecc.). La classe `Shape` dovrebbe prevedere tra gli altri metodi sicuramente un metodo `draw()` che disegna la figura geometrica. Tale metodo sarà poi ridefinito in ogni sottoclasse. Il fatto che il metodo è definito al livello della classe base `Shape` garantisce che esso può essere invocato in modo uniforme relativamente a tutti gli oggetti facenti parte della gerarchia e rappresentanti varie figure geometriche.

Ma la classe base che implementazione dovrebbe dare al metodo `draw()`? Siccome la classe base non rappresenta nessuna figura geometrica specifica non può fornire alcuna implementazione significativa del metodo. Potrebbe semplicemente lasciare l'implementazione vuota: `draw() {}`. Questo può essere una soluzione accettabile se il metodo è come `draw()` che non ritorna nessun valore. Ma se il metodo dovesse ritornare un valore? Quale valore ritorna? Inoltre anche nel caso di metodi che come `draw()` non ritornano valori, la soluzione dell'implementazione vuota non è pienamente soddisfacente perché tende a

nascondere il fatto che la classe base (nel nostro esempio `Shape`) non è una classe *concreta*. Nel senso che gli oggetti di tale classe non possono essere usati direttamente. Gli oggetti della classe `Shape` non rappresentano alcuna figura specifica e quindi non possono in nessun modo essere usati direttamente, solamente gli oggetti delle sottoclassi possono essere usati direttamente. In altre parole, non ha senso istanziare oggetti della classe `Shape`.

Proprio allo scopo di fornire strumenti per risolvere in modo soddisfacente situazioni come quella appena descritta, il linguaggio Java permette di definire *classi astratte* (*abstract classes*). Una classe astratta è come una classe normale (concreta) con però uno o più metodi senza implementazione. Un metodo senza implementazione è un *metodo astratto* (*abstract method*) per il quale è definita solamente l'intestazione (ovvero l'interfaccia). La sintassi per definire metodi astratti e classi astratte è molto semplice. È sufficiente usare il modificatore **`abstract`** e terminare l'intestazione dei metodi astratti con "`;`" che sostituisce il corpo del metodo. Ecco un breve elenco delle caratteristiche principali di una classe astratta.

- Una classe con un metodo astratto deve essere dichiarata astratta.
- Una classe astratta non può essere istanziata.
- Una sottoclasse di una classe astratta può essere istanziata solo se implementa tutti i metodi astratti della superclasse.
- Se una sottoclasse di una classe astratta non implementa tutti i metodi astratti che eredita è essa stessa astratta e deve essere esplicitamente dichiarata astratta.
- Metodi `static` o `private` non possono essere astratti (perché tali metodi non sono ereditati dalle sottoclassi e quindi non sarebbero mai implementati).

Oltre a queste caratteristiche che la differenziano da una classe concreta, una classe astratta è del tutto simile ad una classe normale.

La prima classe - versione 3 Grazie alle classi astratte possiamo ristrutturare le classi `CharRect` e `PrintMedium`. Prima di tutto introdurremo una classe astratta, che chiameremo `CharShape`, che rappresenta una generica figura di caratteri. La classe `CharRect` sarà una delle sottoclassi concrete di `CharShape`. Un'altra sarà `CharPyramid`. Ovviamente se ne possono aggiungere altre a piacimento. D'altronde uno degli scopi della nuova struttura è proprio quello di facilitare l'estensione delle funzionalità del sistema. Inoltre, le classi astratte risultano utili anche per migliorare la classe `PrintMedium`. La classe `PrintMedium` diventerà una classe astratta e per ogni mezzo di stampa specifico si introdurrà una corrispondente sottoclasse concreta di `PrintMedium`.

Per mantenere le classi semplici in modo da focalizzare l'attenzione sulle relazioni tra le classi, implementeremo una versione semplificata della classe `CharRect`. Rispetto all'ultima versione prevediamo un solo carattere e un solo metodo di stampa. Iniziamo dalla definizione della classe `CharShape`:

```
// package in cui sono definite tutte le classi della gerarchia di CharShape
package charshape;

import printmedium.*;    // il package in cui è definita la classe PrintMedium

public abstract class CharShape {
    private static final char DEF_FILLCHAR = '*';

    private char fillChar = DEF_FILLCHAR;
    private int left, top;
    private PrintMedium pMedium;

    public CharShape(PrintMedium pm, int l, int t) {
        left = l;
        top = t;
        pMedium = pm;
    }

    public void setChar(char c) { fillChar = c; }
    public void setPM(PrintMedium pm) { pMedium = pm; }

    public abstract void draw();    // metodi astratti che saranno implementati
    public abstract int area();     // nelle sottoclassi concrete
        // metodo di utilità che stampa una linea di caratteri nella riga r con lo
```

```
    void drawRow(int r, int offset, int length) {   // specificato offset rispetto a
        for (int k = 0 ; k < length ; k++)          // left e di lunghezza length
            pMedium.printChar(top + r, left + offset + k, fillChar);
    }

    void end() { pMedium.end(); }
}
```

Essenzialmente la classe si occupa di gestire la stampa a "basso livello" fornendo un metodo di utilità `drawRow()` che stampa una linea di caratteri che inizia in una specificata posizione di una riga e ha una certa lunghezza. Si noti che i metodi `drawRow()` e `end()` hanno accesso limitato al package `charshape` perché tali metodi servono solamente per l'implementazione delle sottoclassi. Passsiamo ora alla definizione delle sottoclassi.

```
package charshape;

import printmedium.*;    // il package in cui è definita la classe PrintMedium

public class CharRect extends CharShape {
    private int width, height;

    public CharRect(PrintMedium pm, int l, int t, int w, int h) {
        super(pm, l, t);       // invoca il costruttore di CharShape
        width = w;
        height = h;
    }

    public void draw() {             // implementa il metodo astratto
        for (int r = 0 ; r < height ; r++)
            drawRow(r, 0, width);
        end();
    }

    public int area() {             // implementa il metodo astratto
        return width*height;
    }
}
```

Ed ecco anche la definizione della classe `CharPyramid`:

```
package charshape;

import printmedium.*;    // il package in cui è definita la classe PrintMedium

public class CharPyramid extends CharShape {
    private int height;

    public CharPyramid(PrintMedium pm, int l, int t, int h) {
        super(pm, l, t);          // invoca il costruttore di CharShape
        height = h;
    }

    public void draw() {               // implementa il metodo astratto
        for (int r = 0 ; r < height ; r++)
            drawRow(r, height - r - 1, 2*r + 1);
        end();
    }

    public int area() {               // implementa il metodo astratto
        return height*height;
    }
}
```

Grazie al metodo `drawRow()`, le implementazioni dei metodi `draw()` delle due sottoclassi sono particolarmente semplici ed evitano duplicazioni di codice. Veniamo ora alla classe `PrintMedium`.

```
// il package in cui sono definite tutte le classi della gerarchia PrintMedium
package printmedium;
```

```
public abstract class PrintMedium {
    public abstract void printChar(int row, int col, char c);

    public void end() {}
}
```

Siccome per la maggior parte dei mezzi di stampa il metodo `end()` ha una implementazione vuota la classe `PrintMedium` fornisce questa implementazione invece di definire il metodo in modo astratto. E ora definiamo due sottoclassi per i mezzi di stampa relativi alle matrici di caratteri e ai flussi. Iniziamo con le matrici di caratteri:

```
package printmedium;

public class PrintMMatrix extends PrintMedium {
    private char[][] matrix;

    public PrintMMatrix(char[][] m) { matrix = m; }
         // implementa il metodo astratto
    public void printChar(int row, int col, char c) {
        matrix[row][col] = c;
    }
}
```

E poi la sottoclasse per i flussi:

```
package printmedium;

import java.io.*;

public class PrintMStream extends PrintMedium {
    private PrintStream stream;
    private int currRow = 0, currCol = 0;

    public PrintMStream(PrintStream s) { stream = s; }
        // implementa il metodo astratto
    public void printChar(int row, int col, char c) {
        if (currRow < row) currCol = 0;
        for ( ; currRow < row ; currRow++) stream.println();
        for ( ; currCol < col ; currCol++) stream.print(' ');
        stream.print(c);
        currCol++;
    }

    public void end() {    // ridefinisce il metodo di PrintMedium
        stream.println();
        currRow = currCol = 0;
    }
}
```

La sottoclasse deve ridefinire il metodo `end()`. Grazie all'introduzione della classe astratta `PrintMedium` e delle sottoclassi relative ai mezzi di stampa, la struttura complessiva è più chiara e precisa. Nella definizione originale i dettagli dei due mezzi di stampa erano mescolati in un unica classe invece adesso sono ben separati. La nuova struttura rende inoltre più agevole introdurre nuovi mezzi di stampa.

Diamo ora un semplice programma che mette alla prova le nuove classi:

```
import printmedium.*;
import charshape.*;

public class Test {
        // pulisce una matrice di caratteri, implementazione omessa
    private static void clear(char[][] screen) {...}
        // stampa una matrice di caratteri, implementazione omessa
    private static void print(char[][] screen) {...}
    public static void main(String[] args) {
        char[][] screen = new char[12][50];
        clear(screen);
        PrintMMatrix pM = new PrintMMatrix(screen);
        CharShape[] shape = new CharShape[4];
        shape[0] = new CharRect(pM, 2, 0, 6, 4);
```

```
        shape[1] = new CharRect(pM, 10, 2, 5, 5);
        shape[2] = new CharPyramid(pM, 0, 6, 4);
        shape[3] = new CharPyramid(pM, 10, 3, 7);
        shape[1].setChar('o');
        shape[3].setChar('#');
        for (int i = 0 ; i < shape.length ; i++) shape[i].draw();
        print(screen);
        PrintMStream pOut = new PrintMStream(out);
        shape[0].setPM(pOut);
        shape[3].setPM(pOut);
        shape[0].draw();
        shape[3].draw();
    }
}
```

Grazie all'introduzione della classe `CharShape` possiamo trattare in modo uniforme (tramite un array) diverse figure specifiche come rettangoli e piramidi. L'esecuzione del programma produce il seguente risultato:

```
  ******
  ******
  ******  ooooo
  ******  ooooo #
          ooooo###
          oooo#####
   *      ooo#######
  ***       #########
 *****     ###########
*******   #############


  ******
  ******
  ******
  ******



              #
             ###
            #####
           #######
          #########
         ###########
        #############
```

Questi primi esempi mostrano un particolare uso delle classi astratte che certo non esaurisce lo spettro dei possibili usi. Il prossimo esempio mostrerà un uso diverso ma altrettanto importante.

Il *Template* design pattern Dovrebbe essere ormai chiaro che un linguaggio di programmazione orientato agli oggetti come Java offre strumenti sofisticati per la progettazione del software. Ereditarietà, polimorfismo e classi astratte sono strumenti potenti che da una parte possono semplificare la struttura di un programa e dall'altra la rendono più sofisticata e delicata. Il loro buon uso non è affatto scontato. Proprio per suggerire un buon uso di questi strumenti sono stati individuati e studiati molti modi di usare tali strumenti. I migliori di questi modi, o modelli di progettazione, sono stati sistematicamente raccolti e descritti così che siano a disposizione di un qualsiasi programmatore. Questi modelli di progettazione software (tramite linguaggi orientati agli oggetti) sono comunemente chiamati *design patterns*.

Adesso vedremo un esempio di uno di questi design patterns che è particolarmente adatto ad essere realizzato tramite classi astratte. Questo design pattern è chiamato *Template* ed è atto a definire lo scheletro di un algoritmo (o procedura), che deve realizzare una o più operazioni, in cui l'implementazione di alcune parti è demandata alle sottoclassi. Così le sottoclassi implementano alcune parti dell'algoritmo lasciandone inalterata la struttura. Gli usi più comuni del design pattern Template si trovano in librerie che forniscono framework per l'implementazione di applicazioni. In questi casi una classe astratta (fornita dalla libreria) implementa alcune funzionalità generali e comuni a tutte le applicazioni (gestione degli eventi, menu, finestre di dialogo, ecc.) e lascia alle sottoclassi (che corrispondono ad applicazioni concrete) il compito di implementare le specifiche azioni da compiere in

risposta agli eventi (movimenti del mouse, tasti premuti, ecc.) generati dall'utente.

Ovviamente non possiamo qui descrivere un esempio realistico perché sarebbe di gran lunga troppo complicato. Possiamo però descrivere un esempio molto semplificato che mostra comunque il design pattern Template in azione. Definiamo una classe astratta `TextMenuApp` il cui scopo è di fornire lo scheletro per applicazioni (programmi) la cui interazione con l'utente è basata su un menu testuale. Quindi la classe `TextMenuApp` gestisce il menu testuale, le cui voci saranno inizializzate dalla sottoclasse, e invoca un metodo astratto, che sarà implementato dalla sottoclasse, in risposta alle scelte effettuate dall'utente. Ecco la definizione della classe.

```
package menuapp;    // package in cui è definita solamente questa classe

import java.util.*;

public abstract class TextMenuApp {
    private String[] menu;
        // costruttore protetto accessibile solamente alle sottoclassi
    protected TextMenuApp(String...item) {
        int n = item.length;
        menu = new String[n + 1];
        for (int i = 0 ; i < n ; i++)
            menu[i] = item[i];
        menu[n] = "ESCI";
    }
        // questo metodo esegue l'applicazione
    public void run() {
        int n = menu.length;
        Scanner input = new Scanner(in);
        boolean quit = false;
        while (!quit) {
            for (int i = 0 ; i < n ; i++)
                out.println((i+1)+". "+menu[i]);
            int choice = input.nextInt();
            if (choice >= 1 && choice < n) doMenu(choice);
            else if (choice == n) quit = true;
        }
    }
        // metodo astratto che deve essere implementato: esegue il k-esimo menu
    protected abstract void doMenu(int k);
}
```

Le voci del menu sono fornite tramite il costruttore, la voce `"ESCI"` è direttamente implementata dalla classe. Si noti che il costruttore e il metodo astratto `doMenu()` sono definiti usando il modificatore di accesso **`protected`**. Tale modificatore dichiara che l'accesso è ristretto al package (nel nostro caso `menuapp`) e alle sottoclassi, anche se appartenenti a package differenti. Quindi si tratta di una modalità di accesso più ampia di quella di default che limita l'accesso solamente all'interno del package. Qui il modificatore `protected` è usato perchè non ha senso che classi al di fuori del package `menuapp` che non sono sottoclassi di `TextMenuApp` possano accedere al costruttore e al metodo `doMenu()`. Mentre è necessario che l'accesso sia garantito alle sottoclassi anche se non appartengono al package `menuapp`. Infatti, in una realizzazione realistica la classe `TextMenuApp` sarà parte di una libreria di uso generale e quindi le sue classi apparteranno a opportuni package, mentre le sottoclassi clienti che forniscono le implementazioni relative ad applicazioni concrete apparteranno necessariamente a package differenti.

Come esempio di sottoclasse che implementa una applicazione concreta consideriamo una semplicissima classe che realizza una applicazione che permette di calcolare alcune funzioni matematiche scelte dall'utente tramite il menu.

```
import menuapp.*;
import java.util.*;

public class MathApp extends TextMenuApp {
    public MathApp() {
        super("LOGARITMO", "RADICE QUADRATA");  // invoca il costruttore di TextMenuApp
    }
        // implementa il metodo astratto della classe TextMenuApp
    public void doMenu(int choice) {
        Scanner input = new Scanner(in);
        out.print("DIGITA UN NUMERO: ");
```

```
        double x = input.nextDouble();
        switch(choice) {
            case 1: out.println("LOG("+x+") = "+Math.log(x)); break;
            case 2: out.println("SQRT("+x+") = "+Math.sqrt(x)); break;
        }
    }
}
```

Ed ecco un programma che mette alla prova le due classi:

```
public class Test {
    public static void testMathApp() {
        MathApp app = new MathApp();
        app.run();
    }
}
```

Ed infine, ecco una possibile esecuzione del programma:

```
1. LOGARITMO
2. RADICE QUADRATA
3. ESCI
1
DIGITA UN NUMERO: 3
LOG(3.0) = 1.0986122886681096
1. LOGARITMO
2. RADICE QUADRATA
3. ESCI
2
DIGITA UN NUMERO: 34
SQRT(34.0) = 5.830951894845301
1. LOGARITMO
2. RADICE QUADRATA
3. ESCI
3
```

Il design pattern Template si basa su una struttura di controllo in un certo senso invertita perchè una superclasse (`TextMenuApp`) invoca le operazioni (`doMenu()`) implementate in una sottoclasse (`MathApp`). Questo genere di struttura di controllo è anche conosciuta con il nome pittoresco di "the Hollywood principle" cioè "Don't call us, we'll call you".

Esercizi

[Piramidi_obligue] Aggiungere alla gerarchia di classi `CharShape` una sottoclasse `CharObliPyramid` che permette di stampare piramidi obligue secondo un parametro non negativo `obliqueness` che determina di quante posizioni è spostato verso destra ogni livello della piramide (rispetto alla piramide normale). Ecco alcuni esempi relativi a piramidi di altezza 5 e con valori di `obliqueness` differenti:

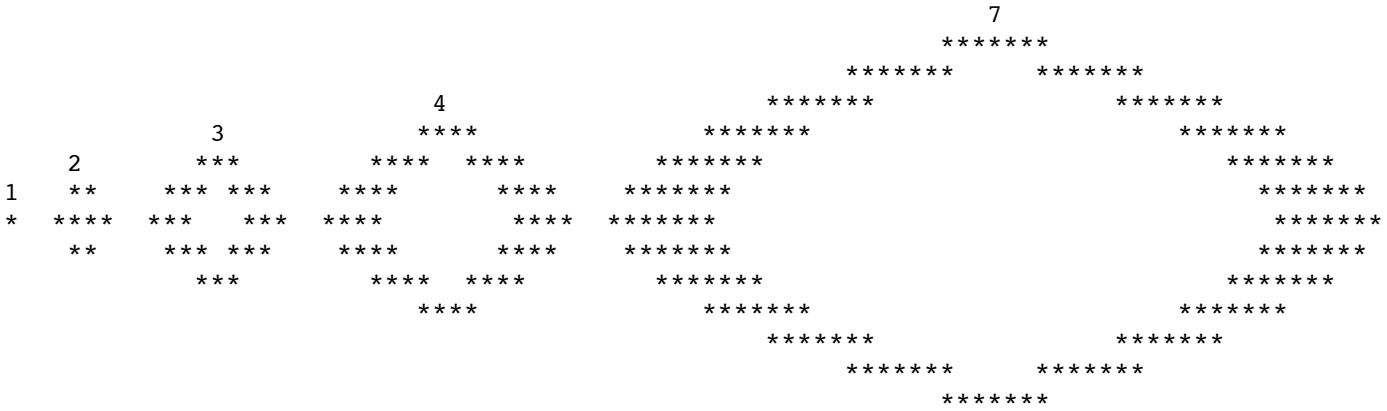
```
obliqueness:       0               1               2                       5
                   *               *               *                       *
                  ***             ***             ***                     ***
                 *****           *****           *****                   *****
                *******         *******         *******                 *******
               *********       *********       *********               *********
```

[Cornici] Aggiungere alla gerarchia di classi `CharShape` una sottoclasse `CharFrame` che permette di stampare delle cornici rettangolari di vario spessore. Lo spessore è determinato da un parametro `thickness` secondo i seguenti esempi (la `thickness` è indicata sopra ogni figura):

```
   1             2                3
  ****       **********       **********
  *  *       **********       **********
  *  *       **      **       **********
  ****       **      **       ***    ***
             **********       **********
             **********       **********
```

```
**********
```

[Ellissi] Aggiungere alla gerarchia di classi `CharShape` una sottoclasse `CharEllipse` che permette di stampare delle forme che ricordano delle ellissi. La forma della figura ellittica che viene stampata dipende da un parametro `width` il cui significato è spiegato dai seguenti esempi (la `width` è indicata sopra ogni figura):



[Strisce_verticali] Modificare la classe `CharShape` e le sue sottoclassi afffinché si possano stampare le figure con una tessitura a strisce verticali come nei seguenti esempi:

```
*o*o*        *          o          #!#!#!       #
*o*o*       o*o        *o*         #!#!#!      !#!
*o*o*      *o*o*      o*o*o                   #!#!#
                     *o*o*o*
```

Suggerimento: Ovviamente la classe `CharShape` dovrà mantenere due caratteri e avere dei metodi per poter modificare tali caratteri (`setChar()`). Conviene modificare il metodo `drawRow()` oppure aggiungere un'altro metodo? La stampa a strisce verticali deve essere fatta tramite uno specifico metodo diverso da `draw()` oppure potrebbe essere fatta sempre tramite il metodo `draw()`. In quest'ultimo caso le sottoclassi che rappresentano le figure concrete dovrebbero anche specificare la tessitura (normale o a strisce verticali).

[Collisione2] Aggiungere alla gerarchia di `PrintMedium` una sottoclasse per gestire il mezzo di stampa specificato nell'esercizio [Collisione].

[Menu] Modificare la classe `TextMenuApp` in modo tale che permetta di visualizzare il menu con una cornice come nell'esempio qui sotto:

```
************************
*  1. LOGARITMO        *
*  2. RADICE QUADRATA  *
*  3. ESCI             *
************************
```

Però la sottoclasse deve poter decidere se usufruire o meno di questa opzione.

[File_app] Definire una sottoclasse `FileApp` di `TextMenuApp` che gestisce una applicazione con il seguente menu:

```
1. IMPOSTA FILE
2. INFO
3. STAMPA
4. CERCA
5. ESCI
```

L'operazione "IMPOSTA FILE" permette all'utente di digitare il nome d un file (che diventerà il file impostato sul quale agiranno le altre operazioni). L'operazione "INFO" stampa le informazioni relative al file impostato (pathname assoluto, lunghezza, ecc.). L'operazione "STAMPA" stampa a video il contenuto del file. L'operazione "CERCA" permette all'utente di digitare una stringa e poi la cerca nel file impostato e l'esito della ricerca è riportato a video.

Interfacce

Quando una sottoclasse eredita un metodo della superclasse ne eredita l'interfaccia (cioè l'intestazione

del metodo) e, se non lo ridefinisce, anche l'implementazione. Gli esempi relativi alle classi astratte mostrano che a volte l'ereditarietà dell'implementazione non è necessaria e anzi può diventare un inutile fardello. I metodi astratti permettono di venire incontro proprio a questa esigenza di avere solamente una ereditarietà di interfaccia. Questo non è casuale perché è proprio l'ereditarietà di interfaccia che abilita l'uso di una delle caratteristiche più utili dell'ereditarietà: il polimorfismo.

Il termine interfaccia è comunemente usato anche con un significato più stringente e però anche meno formale. Infatti, di solito per interfaccia di un metodo si intende l'intestazione del metodo insieme con la specifica del risultato che deve essere prodotto dall'invocazione del metodo. Ad esempio, l'interfaccia del metodo astratto `void printChar(int row, int col, char c)` (della classe `PrintMedium`) non solo definisce una intestazione, che sarà automaticamente ereditata dalle sottoclassi, ma specifica informalmente anche quale deve essere il risultato della sua invocazione (cioè, la stampa del carattere `c` nella posizione `(i, j)` del mezzo di stampa rappresentato dall'oggetto). Quindi una interfaccia, nella sua accezione più stringente, consiste di una parte sintattica, le intestazioni dei metodi, e di una parte semantica, le specifiche dei risultati delle invocazioni dei metodi.

Purtroppo il termine interfaccia è usato in modo ambiguo. A volte lo si usa per intendere solamente la parte sintattica e altre volte per intendere entrambi le parti, quella sintattica insieme a quella semantica. Per questa ragione è stato introdotto un termine specifico per indicare sia la parte sintattica che quella semantica: il *contratto*. Una classe che implementa una interfaccia è come se aderisse a un contratto: si obbliga a rispettare le intestazioni dei metodi e per ognuno di essi si obbliga a produrre, a seguito di una invocazione, il risultato richiesto. Ovviamente, solamente la parte sintattica del contratto può essere gestita automaticamente. La parte semantica è una responsabilità del programmatore.

Il linguaggio Java offre un meccanismo più flessibile e affidabile per l'ereditarietà di interfaccia di quello offerto dalla ereditarietà tra classi che abbiamo già visto. È possibile definire una interfaccia, tramite la parola chiave **`interface`**, in cui si definiscono solamente le intestazioni dei metodi. Una **`interface`** è quindi simile ad una classe astratta in cui tutti i metodi sono astratti (e non ci sono costruttori). Però c'è una differenza fondamentale: le **`interface`** supportano l'ereditarietà multipla. Questo significa che una classe può implementare più di una interfaccia e una interfaccia può estendere più interfacce. Come vedremo questa maggiore flessibilità risulta molto utile e infatti è frequentemente usata nelle librerie di Java.

La definizione di una interfaccia è simile a quella di una classe con la parola chiave `interface` al posto della parola chiave `class`. Però ci sono delle regole speciali per i membri di una interfaccia. Ecco un elenco delle principali regole e caratteristiche delle interfacce.

- Una interfaccia può avere tre tipi di membri: metodi, costanti, classi e interfacce nidificate. Non può definire costruttori e una interfaccia non può essere istanziata.
- Tutti i membri di una interfaccia sono implicitamente `public` e per convenzione il modificatore `public` è omesso.
- Le costanti sono definite come campi che sono implicitamente `public`, `static` e `final`, per convenzione questi modificatori sono omessi.
- I metodi sono implicitamente `public` e `abstract` e per convenzione questi modificatori sono omessi. Nessun'altro modificatore è permesso, in particolare, i metodi non possono essere `static`.
- Le classi e interfacce nidificate sono implicitamente `public` e `static`.
- Una interfaccia può estendere, tramite la parola chiave `extends`, una o più interfacce.
- Una interfaccia, al pari di una classe, definisce un tipo appartenente alla famiglia dei tipi riferimento.
- Una classe può, tramite la parola chiave `implements`, dichiarare di implementare una o più interfacce. Questo significa che la classe per ogni metodo dichiarato in queste interfacce deve fornire una implementazione o dichiararlo come `abstract`.

Vedremo parecchi esempi che illustrano vari usi delle interfacce. Iniziamo con una interfaccia per insiemi di stringhe. Consideriamo una interfaccia che definisce le operazioni più comuni relativamente ad un insieme di stringhe. Queste dovrebbero essere: inserimento di una nuova stringa nell'insieme, rimozione di una stringa dall'insieme, determinare se una data stringa è nell'insieme, ecc. Ma, perché vogliamo definire una interfaccia invece che una classe? Il nostro intento potrebbe essere di produrre codice che può essere riusato, come quello di una libreria di uso generale. Se usassimo una classe per rappresentare gli insiemi di stringhe legheremmo indissolubilmente l'implementazione all'interfaccia. Se volessimo usare in una stessa applicazione un'altra implementazione dovremmo definire un'altra classe che necessariamente introdurrà un'altra interfaccia (ovvero un'altro tipo). Quindi le due classi non

potranno essere usate interscambiabilmente, cioè il codice che le usa dovrà esplicitamente trattarle in modo differente non potendo usufrire del polimorfismo. In realtà, per risolvere questo problema potremmo introdurre una superclasse astratta. Però questa non è sempre la soluzione ottimale. Prima di tutto, potrebbe non esserci codice condivisibile al livello della superclasse e così la superclasse sarebbe priva di qualsiasi implementazione. Allora conviene che ciò sia esplicitato tramite l'uso di una interfaccia, invece di una classe astratta. Inoltre, c'è una ragione molto più importante a favore dell'uso dell'interfacce. Le interfacce supportano l'ereditarietà multipla. Così è possibile, nel caso degli insiemi di stringhe, partizionare le operazioni in gruppi. Ad esempio, in alcune situazioni potrebbero essere di interesse solamente le operazioni che permettono di "leggere" l'insieme (sapere se una data stringa appartiene all'insieme, conoscere la cardinalità dell'insieme, ecc.) e non le operazioni che permettono di modificarlo. Anzi, potrebbe non essere proprio possibile modificare l'insieme perché è mantenuto in un file accessibile in sola lettura. Quindi la nostra interfaccia può essere decomposta in più interfacce: una per le operazioni di sola lettura, un'altra per quelle che modificano l'insieme e magari altre ancora per ulteriori operazioni. Le classi potranno così decidere di implementare solamente quelle interfacce che sono di interesse e non le altre. L'ereditarietà multipla garantirà la possibilità di usufruire del polimorfismo e quindi l'interscambiabilità degli oggetti delle classi ovunque ciò abbia senso.

La nostra prima interfaccia riguarda le operazioni di sola "lettura" dell'insieme di stringhe. Per semplicità ci limitiamo a considerarne solamente due: la ricerca di una stringa nell'insieme e la cardinalità.

```
public interface StrSet {
        // ritorna true se s appartiene all'insieme
    boolean contains(String s);
        // ritorna il numero di stringhe dell'insieme
    int size();
}
```

Come già menzionato, un'interfaccia non definisce solamente le intestazioni dei metodi ma specifica anche quale deve essere il risultato della loro invocazione. Ovvero, definisce un contratto che qualsiasi classe che implementa l'interfaccia si impegna a rispettare. Ovviamente, la specifica del risultato non può essere formalizzata, a differenza dell'intestazione, perciò è descritta a parole nei commenti.

Consideriamo ora due possibili classi che implementano l'interfaccia `StrSet`. La prima rappresenta un insieme di keywords.

```
public class KeywordSet implements StrSet {
    private String[] keywords;
        // costruisce un insieme di keywords copiandole dall'array kA
    public KeywordSet(String...kA) {  // l'array deve essere clonato per evitare
        keywords = kA.clone();        // che modificando l'array fornito in input kA
    }                                 // si modifichi anche questo insieme
        // implementa il metodo dell'interfaccia StrSet
    public boolean contains(String s) {
        for (int i = 0 ; i < keywords.length ; i++)
            if (keywords[i].equals(s)) return true;
        return false;
    }
        // implementa il metodo dell'interfaccia StrSet
    public int size() { return keywords.length; }
}
```

Il costruttore usa il metodo `clone()` per clonare, ovvero creare una copia, dell'array. Questo è un metodo della classe `Object` che gli array ridefiniscono. Discuteremo i dettagli di questo metodo più avanti quando avremo parlato della genericità. La seconda implementazione riguarda un dizionario di stringhe o parole.

```
import java.io.*;
import java.util.*;

public class Dictionary implements StrSet {
    private File file;
    private Scanner scan;
    private int size = -1;

    public Dictionary(String pathname) throws FileNotFoundException {
        file = new File(pathname);
```

```
        scan = new Scanner(file);
    }
        // riporta il cursore all'inizio del file
    public void rewind() throws FileNotFoundException {
        scan.close();
        scan = new Scanner(file);
    }
        // implementa il metodo dell'interfaccia StrSet
    public boolean contains(String s) {
        try {                   // non può usare la dichiarazione throws perché
            rewind();           // questa non è prevista dall'interfaccia
        } catch (FileNotFoundException ex) { return false; }
        while (scan.hasNext())
            if (scan.next().equals(s)) return true;
        return false;
    }
        // implementa il metodo dell'interfaccia StrSet
    public int size() {
        if (size == -1) {
             try {               // non può usare la dichiarazione throws perché
                 rewind();       // questa non è prevista dall'interfaccia
             } catch (FileNotFoundException ex) { return 0; }
             size = 0;
             while (scan.hasNext()) {
                 size++;
                 scan.next();
             }
        }
        return size;
    }
        // ritorna la prossima stringa, se non c'è ritorna null
    public String next() {
        if (scan.hasNext()) return scan.next();
        else return null;
    }
}
```

Si osservi che l'implementazione dei metodi dell'interfaccia `StrSet` non può usare la dichiarazione `throws` (per dichiarare il possibile lancio dell'eccezione controllata `FileNotFoundException`) perché ciò non è esplicitamente dichiarato nell'interfaccia. Vediamo ora un semplice programma che usa le due classi e quindi anche l'interfaccia.

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

public class Test {
    public static boolean searchWord(StrSet[] ss, String word) {
        for (int i = 0 ; i < ss.length ; i++)
            if (ss[i].contains(word)) return true;
        return false;
    }
    public static void main(String[] args) throws FileNotFoundException {
        String word = args[0];
        StrSet[] ws = new StrSet[2];
        ws[0] = new KeywordSet("Musica", "Film", "Teatro", "Cinema");
        ws[1] = new Dictionary("words.txt");
        if (searchWord(ws, word)) out.println("La parola "+word+" e' stata trovata");
        else out.println("La parola "+word+" NON e' stata trovata");
    }
}
```

Come si vede, l'interfaccia `StrSet` può essere usata alla stregua di un qualsiasi tipo riferimento. Il metodo `searchWord()` prende in input un array di `StrSet` e gli oggetti di questo array possono appartenere ad una qualsiasi classe che implementa l'interfaccia `StrSet`. Adesso vedremo alcuni esempi di ereditarietà multipla.

Ereditarietà multipla In alcuni casi l'insieme di stringhe è ordinato. Allora può essere utile poter scandire tutte le stringhe dell'insieme secondo l'ordinamento o meglio ancora poter accedere alla stringa

in una data posizione rispetto all'ordinamento. Per questo introduciamo la seguente interfaccia.

```
public interface SortedStrSet extends StrSet {
        // ritorna la k-esima stringa dell'insieme ordinato. Se k < 0 oppure
    String getKth(int k);    // k > size - 1 ritorna null
}
```

L'interfaccia `SortedStrSet` estende l'interfaccia `StrSet` e quindi ne eredita tutti i metodi. Consideriamo due classi che implementano questa interfaccia estesa. La prima è una classe che rappresenta un insieme ordinato di keywords.

```
import java.util.*;

public class SortedKeywordSet implements SortedStrSet {
    private String[] keywords;

    public SortedKeywordSet(String[] kA) {
        keywords = kA.clone();
        Arrays.sort(keywords);    // ordina l'array in senso ascendente rispetto
    }                             // all'ordine lessicografico prodotto dal metodo
                                  // compareTo() di String
        // implementa il metodo di SortedStrSet
    public boolean contains(String s) {
        return (Arrays.binarySearch(keywords, s) >= 0);    // usa la ricerca binaria
    }                                                      // per cercare la stringa
        // implementa il metodo di SortedStrSet
    public int size() { return keywords.length; }
        // implementa il metodo di SortedStrSet
    public String getKth(int k) {
        if (k < 0 || k > keywords.length - 1) return null;
        return keywords[k];
    }
}
```

Il costruttore usa il metodo `sort()` della classe `Arrays` del package `Java.util` per ordinare l'array. La classe `Arrays` contiene molti metodi statici di utilità per gli array. L'implementazione del metodo `contains()` usa infatti un'altro metodo della classe `Arrays`, `binarySearch()`, che esegue una ricerca binaria per cercare una stringa in un array ordinato.

La seconda classe è una estensione della classe `Dictionary` e gestisce un insieme ordinato di stringhe mantenuto in un file.

```
import java.io.*;

public class SortedDictionary extends Dictionary implements SortedStrSet {
        // si assume che il file contenga le stringhe già ordinate
    public SortedDictionary(String pathname) throws FileNotFoundException {
        super(pathname);
    }
        // implementa il metodo di SortedStrSet
    public String getKth(int k) {
        if (k < 0 || k > size() - 1) return null;
        try {              // non può usare la dichiarazione throws perché
            rewind();      // questa non è prevista dall'interfaccia
        } catch (FileNotFoundException ex) { return null; }
        for ( ; k > 0 ; k--) next();
        return next();
    }
}
```

Questo è un primo esempio di ereditarietà multipla. La classe `SortedDictionary` è sia un sottotipo di `Dictionary` che di `SortedStrSet`. Si osservi che la classe non deve implementare i metodi `contains()` e `size()` perché questi sono già implementati nella superclasse `Dictionary` e la classe `SortedDictionary` li eredita. Ovviamente, se ce ne fosse stato bisogno la classe `SortedDictionary` avrebbe potuto ridefinirli. Si osservi che né `Dictionary` è un sottotipo di `SortedStrSet` né `SortedStrSet` è un sottotipo di `Dictionary`, però sono entrambi supertipi di `SortedDictionary`. In altre parole, ovunque si può usare un oggetto di tipo `Dictionary` e ovunque si può usare un oggetto di tipo `SortedStrSet` si può anche usare un oggetto di tipo `SortedDictionary`.

Vedremo altri esempi di ereditarietà multipla in concomitanza con interfacce relative alle operazioni che modificano un insieme di stringhe.

Liste Introduciamo una interfaccia che estende l'interfaccia `StrSet` con metodi per le due operazioni fondamentali di inserimento e rimozione di stringhe.

```
public interface ModifiableStrSet extends StrSet {
        // aggiunge la stringa s all'insieme, se non è già presente, e ritorna true
    boolean add(String s);    // altrimenti non modifica l'insieme e ritorna false
        // rimuove la stringa s dall'insieme, se è presente, e ritorna true
    boolean remove(String s);    // altrimenti non modifica l'insieme e ritorna false
}
```

Consideriamo due implementazioni di uso generale. La prima tramite array e la seconda tramite liste.

```
import java.util.*;

public class ArrayStrSet implements ModifiableStrSet {
    private String[] array;    // array che mantiene l'insieme di stringhe
    private int count;         // numero di stringhe dell'insieme
        // costruisce un insieme con le stringhe date in input
    public ArrayStrSet(String...sA) {
        array = sA.clone();
        count = array.length;
    }
        // metodo ausiliario che ritorna l'indice in cui è mantenuta la stringa s
    private int find(String s) {               // se non è presente ritorna -1
        for (int i = 0 ; i < count ; i++)
            if (array[i].equals(s)) return i;
        return -1;
    }
        // implementa il metodo dell'interfaccia ModifiableStrSet
    public boolean add(String s) {
        if (find(s) == -1) {
            if (array.length == count)     // se l'array è pieno...
                array = Arrays.copyOf(array, (3*(count + 1))/2);
            array[count++] = s;
            return true;
        } else return false;
    }
        // implementa il metodo dell'interfaccia ModifiableStrSet
    public boolean remove(String s) {
        int k = find(s);
        if (k >= 0) {
            array[k] = array[count - 1];
            count--;
            return true;
        } else return false;
    }
        // implementa il metodo dell'interfaccia ModifiableStrSet
    public boolean contains(String s) {
        return (find(s) >= 0);
    }
        // implementa il metodo dell'interfaccia ModifiableStrSet
    public int size() { return count; }
}
```

Nel metodo `add()` abbiamo usato il metodo `copyOf()` della classe `Arrays` per espandere l'array delle stringhe. Passiamo ora all'implementazione tramite liste. Le liste in Java possono essere implementate in modo molto simile a come lo sono in C. Basterà definire una classe per rappresentare gli elementi della lista in modo analogo a come in C si sarebbe usata una `struct`. Siccome gli elementi della lista sono un dettaglio implementativo che interessa solamente questa classe, la classe che li rappresenta è definita come una classe privata statica.

```
public class LinkedStrSet implements ModifiableStrSet {
        // classe nidificata statica per rappresentare gli elementi della lista
    private static class Elem {
        private String str;     // stringa dell'elemento
```

```
        private Elem next;      // riferimento al prossimo elemento della lista

        private Elem(String s, Elem n) {
            str = s;
            next = n;
        }
    }

    private Elem head;    // mantiene il primo elemento (testa) della lista
        // metodo ausiliario che crea ed aggiunge un nuovo elemento in testa alla lista
    private void addElem(String s) {
        head = new Elem(s, head);
    }
        // costruisce un insieme con le stringhe date
    public LinkedStrSet(String...sA) {
        head = null;
        for (int i = 0 ; i < sA.length ; i++)
            addElem(sA[i]);
    }
        // implementa il metodo dell'interfaccia ModifiableStrSet
    public boolean add(String s) {
        if (!contains(s)) {
            addElem(s);
            return true;
        } else return false;
    }
        // implementa il metodo dell'interfaccia ModifiableStrSet
    public boolean remove(String s) {
        Elem prev = null, p = head;    // scorri la lista mantenendo in prev il
        while (p != null && !p.str.equals(s)) {    // riferimento all'elemento precedente
            prev = p;
            p = p.next;
        }
        if (p != null) {                // se la stringa è presente...
            if (prev == null) head = head.next;    // sgancia l'elemento dalla lista
            else prev.next = p.next;
            return true;
        } else return false;
    }
        // implementa il metodo dell'interfaccia ModifiableStrSet
    public boolean contains(String s) {
        Elem p = head;
        while (p != null && !p.str.equals(s)) p = p.next;
        return (p != null);
    }
        // implementa il metodo dell'interfaccia ModifiableStrSet
    public int size() {
        int count = 0;
        Elem p = head;
        while (p != null) {
            count++;
            p = p.next;
        }
        return count;
    }
}
```

In questo caso una interfaccia, `ModifiableStrSet`, è stata usata per definire un tipo di uso generale, insiemi dinamici di stringhe con le loro operazioni fondamentali, che può avere diverse implementazioni. Ogni implementazione ha i suoi vantaggi e svantaggi. In alcune situazioni può essere preferibile usarne una e in altre situazione è preferibile usare un'altra implementazione. Le librerie di Java contengono molti esempi di questo genere.

Consideriamo un programma che sfrutta l'implementazione fornita per gli insiemi di stringhe per leggere le parole in un file di testo e contare quante parole distinte ci sono e quante parole non compaiono in un elenco di parole contenuto anch'esso in un file. I pathname dei due file sono letti dalla linea di comando. Il file potrebbe essere un romanzo e l'elenco di parole potrebbe contenere le parole di un dizionario. Così il programma stamperebbe il numero totale di parole (comprese le ripetizioni) del romanzo, il numero di parole distinte e il numero di parole (distinte) che non compaiono nel dizionario.

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

public class Test {
    public static void main(String[] args) throws FileNotFoundException {
        String textFile = args[0], wordFile = args[1];
        ModifiableStrSet textSet = new ArrayStrSet();
        Scanner textScan = new Scanner(new File(textFile));
        textScan.useDelimiter("[^\\p{Alpha}]+");
        int count = 0;
        while (textScan.hasNext()) {           // aggiungi le parole del file di testo
            textSet.add(textScan.next());      // all'insieme
            count++;
        }
        out.println("Numero parole: "+count+" di cui distinte: "+textSet.size());
        Scanner wordScan = new Scanner(new File(wordFile));
        while (wordScan.hasNext())              // rimuovi tutte le parole dell'elenco
            textSet.remove(wordScan.next());    // dall'insieme
        out.println("Numero parole non appartenenti al dizionario: "+textSet.size());
    }
}
```

Se invece dell'implementazione tramite array si vuole usare quella basata sulle liste è sufficiente sostituire la linea `ModifiableStrSet textSet = new ArrayStrSet();` con `ModifiableStrSet textSet = new LinkedStrSet();`.

Consideriamo ora una interfaccia per insiemi di stringhe modificabili e ordinati. Potremmo non definire tale interfaccia in quanto abbiamo già le interfacce `ModifiableStrSet` e `SortedStrSet`. Una classe per rappresentare insiemi di stringhe modificabili e ordinati può semplicemente implementare sia l'una che l'altra interfaccia. Ad esempio, potremmo definire una classe del seguente tipo:

```
public class ModSortedStrSet implements ModifiableStrSet, SortedStrSet {
    ...
}
```

Però se prevediamo che in alcune situazioni ci può essere utile avere un tipo che rappresenta proprio insiemi di stringhe modificabili e ordinati, cioè un tipo che supporta sia le operazioni dell'interfaccia `ModifiableStrSet` che quelle dell'interfaccia `SortedStrSet`, allora conviene definire una interfaccia che unisce le due interfacce:

```
public interface ModifiableSortedStrSet extends ModifiableStrSet, SortedStrSet { }
```

L'interfaccia `ModifiableSortedStrSet` non introduce nuovi metodi ma semplicemente unisce i metodi delle due interfacce. Così una classe che implementa l'interfaccia `ModifiableSortedStrSet` non solo è un sottotipo di `ModifiableStrSet` e `SortedStrSet`, come la classe `ModSortedStrSet` ma in più è un sottotipo di `ModifiableSortedStrSet`. Quindi, gli oggetti di tale classe possono essere usati dove è esplicitamente richiesta l'adesione ad entrambe le interfacce. E questo non è possibile per gli oggetti della classe `ModSortedStrSet`.

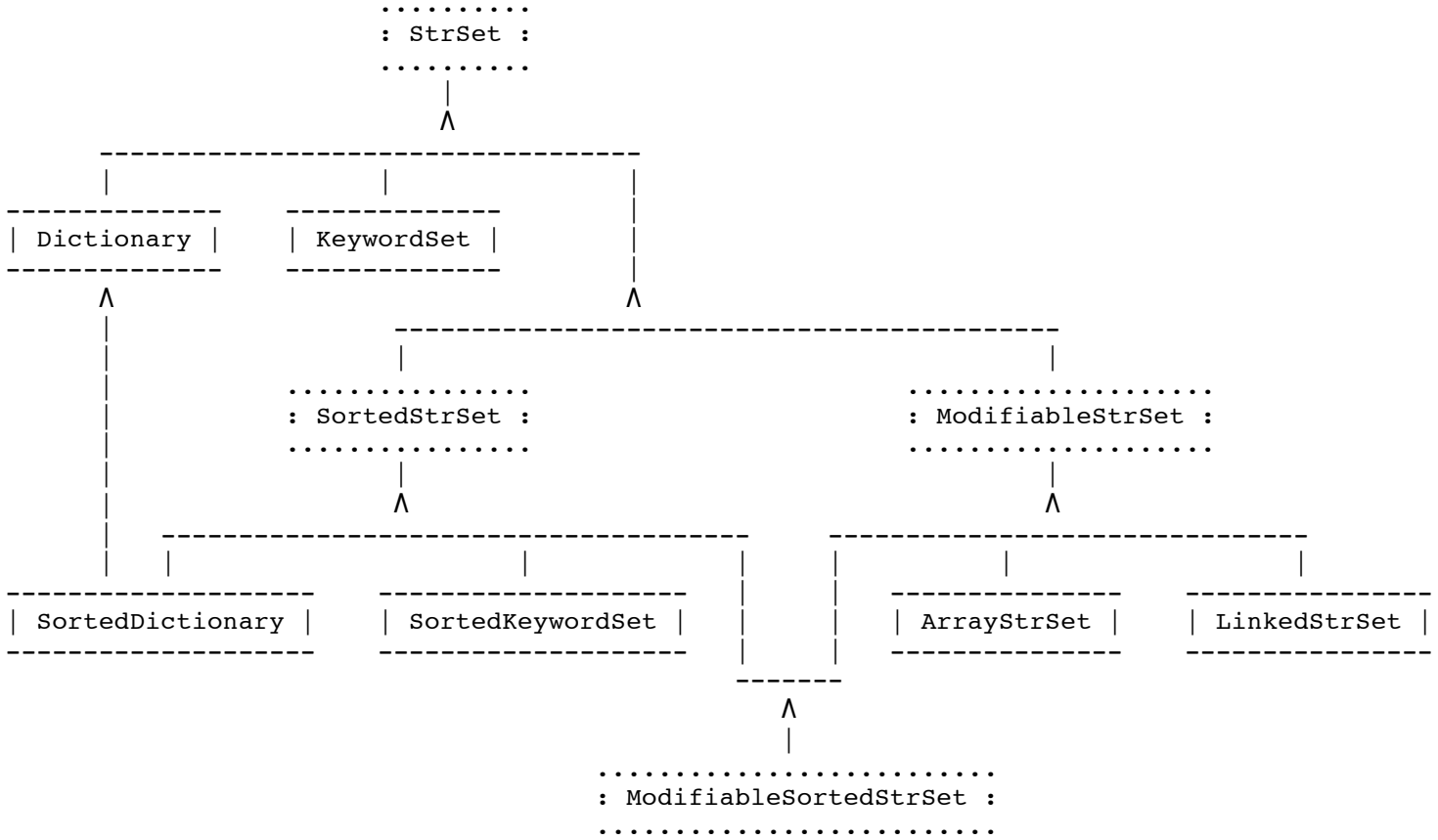
Vediamo ora una implementazione tramite array dell'interfaccia `ModifiableSortedStrSet`. La classe non estende `ArrayStrSet` perchè ciò richiederebbe delle modifiche della classe `ArrayStrSet` e non ci sarebbe comunque molto codice da condividere.

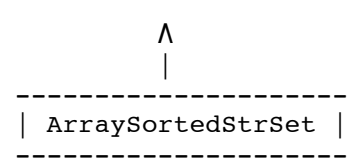
```
import java.util.*;
import static java.util.Arrays.*;

public class ArraySortedStrSet implements ModifiableSortedStrSet {
    private String[] array;    // array che mantiene l'insieme di stringhe
    private int count;         // numero di stringhe dell'insieme
        // costruisce un insieme di stringhe vuoto
    public ArraySortedStrSet() {
        array = new String[0];
        count = 0;
    }
        // implementa il metodo dell'interfaccia ModifiableSortedStrSet
    public boolean add(String s) {
        int j = binarySearch(array, 0, count, s);
```

```
        if (j < 0) {
            j = -(j + 1);
            if (array.length == count)    // se l'array è pieno...
                array = copyOf(array, (3*(count + 1))/2);
            System.arraycopy(array, j, array, j + 1, count - j);
            array[j] = s;
            count++;
            return true;
        } else return false;
    }
        // implementa il metodo dell'interfaccia ModifiableSortedStrSet
    public boolean remove(String s) {
        int j = binarySearch(array, 0, count, s);
        if (j >= 0) {
            System.arraycopy(array, j, array, j - 1, count - j - 1);
            count--;
            return true;
        } else return false;
    }
        // implementa il metodo dell'interfaccia ModifiableSortedStrSet
    public boolean contains(String s) {
        return (binarySearch(array, 0, count, s) >= 0);
    }
        // implementa il metodo dell'interfaccia ModifiableSortedStrSet
    public int size() { return count; }
        // implementa il metodo dell'interfaccia ModifiableSortedStrSet
    public String getKth(int k) {
        if (k < 0 || k >= count) return null;
        return array[k];
    }
}
```

Siccome questa classe è un sottotipo di `ModifiableStrSet` può essere usata nel programma precedente per l'analisi di un file di testo. Se si confrontano i tempi di esecuzione (usando file di input abbastanza grandi) delle versioni con `ArrayStrSet`, `LinkedStrSet` e `ArraySortedStrSet` ci si accorgerà che la versione che usa `ArraySortedStrSet` è almeno dieci volte più veloce delle altre due.

Le interfacce e le classi che abbiamo definito formano una gerarchia. Però non è una gerarchia pura, come si vede dal seguente diagramma:





Le interfacce sono usate in molti modi diversi. Per ora abbiamo toccato solamente alcuni di questi usi. Ne vedremo molti altri dopo aver introdotto la genericità. Infatti l'accoppiata interfacce-genericità è una delle più riuscite ed utili.

Esercizi

[Liste_ordinate] Definire una classe `LinkedSortedStrSet` che implementa l'interfaccia `ModifiableSortedStrSet` tramite liste.
[Ricerca_prefissi] Definire una interfaccia `PrefixSearchable` che estende `StrSet` e introduce un solo metodo `String[] searchPrefix(String p)` che ritorna in un array tutte le stringhe dell'insieme che hanno il prefisso `p`. Modificare le classi `Dictionary` e `LinkedStrSet` in modo che implementino anche tale interfaccia.
[Insiemi_di_interi] Definire delle interfacce `IntSet`, `SortedIntSet` e `ModifiableIntSet` in modo analogo alle interfacce `StrSet`, `SortedStrSet` e `ModifiableStrSet` ma per insiemi di interi (`int`) invece che di stringhe. Definire anche una implementazione di `ModifiableIntSet` tramite liste.
[Insiemi_di_oggetti] Definire delle interfacce analoghe a `StrSet` e `ModifiableStrSet` per insiemi di `Object`. Come uguaglianza si usi l'identità, cioè, due oggetti sono uguali solo se sono lo stesso oggetto. Implementare le interfacce tramite liste e tramite array.
[Sequenze] Definire una interfaccia `ObjSeq` per sequenze di oggetti. Una sequenza di oggetti è un elenco di oggetti, anche ripetuti, con un ordine dato da come gli oggetti sono stati inseriti nell'elenco. L'interfaccia deve definire i seguenti metodi:

`void add(int pos, Object elem)`
: Inserisce l'elemento `elem` nella posizione `pos` spostando di una posizione gli elementi dalla posizione `pos` in poi. La prima posizione è 0. Se `pos < 0` o `pos > n - 1` (dove `n` è il numero di elementi presenti), lancia l'eccezione `IllegalArgumentException`.

`Object get(int pos)`
: Ritorna l'elemento in posizione `pos`. Se `pos < 0` o `pos > n - 1` (dove `n` è il numero di elementi presenti), lancia l'eccezione `IllegalArgumentException`.

`Object remove(int pos)`
: Rimuove dalla sequenza l'elemento in posizione `pos` e scala di una posizione tutti gli eventuali elementi successivi. Ritorna l'elemento rimosso. Se `pos < 0` o `pos > n - 1` (dove `n` è il numero di elementi presenti), lancia l'eccezione `IllegalArgumentException`.

`boolean remove(Object x)`
: Rimuove la prima occorrenza dell'oggetto `x` (l'uguaglianza è determinata dall'invocazione del metodo `equals()`) e ritorna `true`. Se non ci sono oggetti uguali a `x` ritorna `false`.

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

Fornire anche una implementazione tramite liste dell'interfaccia `ObjSeq`.
[Code_stringhe] Definire una interfaccia `StrQueue` per code di `String`. Fornire delle implementazioni tramite array e tramite liste.
[Code_oggetti] Definire una interfaccia `ObjQueue` per code di `Object`. Fornire delle implementazioni tramite array e tramite liste.
[Pile_oggetti] Definire una interfaccia `ObjStack` per pile di `Object`. Fornire delle implementazioni tramite array e liste. Inoltre usare le implementazioni di `ObjStack` per scrivere un metodo che presa in input una stringa che può contenere le parentesi `(`, `)`, `[`, `]`, `{`, `}`, determina se la disposizione delle parentesi è corretta, cioè ogni parentesi aperta è accoppiata ad una parentesi chiusa dello stesso tipo. Ad esempio la stringa `"a+{(b-c)*[a-b]}"` è corretta mentre la stringa `"a+{(b-c)*[a-b]]"` non è corretta.
[Parole_connesse] Diciamo che due parole (stringhe) sono *adiacenti* se hanno la stessa lunghezza e differiscono in esattamente un carattere. Ad esempio, `"porta"` e `"torta"` sono adiacenti così come `"privato"` e `"provato"`, ma `"torta"` e `"trota"` non sono adiacenti. Dato un elenco di parole D, diciamo che due parole u e v di D sono *connesse* rispetto a D se esistono parole w_1, w_2,..., w_k in D tali

che $w_1 = u$, $w_k = v$ e w_i è adiacente a w_{i+1}, per ogni $i = 1,2,...,k-1$. Scrivere un programma che legge in input il nome di un file che contiene un elenco di parole D e una parola u e calcola il numero di parole che sono connesse rispetto a D alla parola u.

Suggerimento: Usare una delle implementazioni di `StrSet` per mantenere D. Poi, usare una delle implementazioni di `ModifiableStrSet` per mantenere l'insieme delle parole connesse ad u finora trovate e usare una delle implementazioni di `StrQueue` (vedi esercizio [Code_stringhe]) per mantenere le parole da cui è ancora possibile derivare parole connesse ad u. Per provare il programma si può usare il seguente elenco di parole: http://www.gilda.it/giochidiparole/elenchi/abc.zip.

[Cammini_di_parole] Scrivere un programma che prende in input il nome di un file che contiene un elenco di parole D, due parole u e v di D e stampa, se esiste, il cammino di parole più corto che connette (rispetto a D) u a v (si vedano le definizioni date nell'esercizio precedente). Ad esempio, se l'elenco di parole è http://www.gilda.it/giochidiparole/elenchi/abc.zip e le parole sono `"privato"` e `"oggetto"` un possibile cammino di parole è:

```
privato
privano
provano
piovano
piovane
giovane
giovate
gioiate
giriate
tiriate
turiate
turiste
turisti
turismi
turismo
aurismo
autismo
autismi
autisti
artisti
ardisti
ardesti
arresti
arresto
agresto
agretto
aggetto
oggetto
```

Suggerimento: Usare una delle implementazioni di `StrSet` per mantenere D. Usare una delle implementazioni di `ModifiableStrSet` per mantenere l'insieme delle parole connesse ad u finora trovate. Poi definire una classe per oggetti che mantengono una parola w e il riferimento all'oggetto dello stesso tipo che contiene la parola dalla quale la parola w è stata derivata (tramite adiacenza). Usare una delle implementazioni di `ObjQueue` (vedi esercizio [Code_oggetti]) per mantenere la coda degli oggetti del tipo suddetto da cui è ancora possibile derivare parole connesse ad u (fino a quando non viene derivata la parola v o la coda si svuota).

[Cammini_di_parole+] Modificare il programma dell'esercizio precedente adoperando una diversa definizione di adiacenza: due parole u e v (anche di lunghezze differenti) sono *adiacenti* se v può essere ottenuta da u o sostituendo un carattere con un'altro o eliminando un carattere o inserendo un nuovo carattere. Ad esempio, le seguenti coppie di parole sono adiacenti secondo questa nuova definizione: `arresti-agresti`, `arresti-arresi`, `caso-casto`.