Realizzazione di record logici e tabelle

Il Java realizza questi tipi di dato strutturati mediante classi create appositamente dal programmatore. Ricordiamo che un record logico è un insieme di elementi, in genere di tipo diverso, denominati campi. Per realizzare ADT record logico si può utilizzare una classe i cui attributi privati sono i campi descritti nel tracciato del record stesso. L'accesso agli attributi/campi della classe/record avviene in modo controllato dal programmatore attraverso un costruttore con parametri e/o dei metodi di accesso.

Supponiamo di voler realizzare il record logico *Studente* definito, per semplicità, soltanto dai campi: cognome, nome, classe e indirizzo. Per questo obiettivo possiamo pensare al seguente schema iniziale di codice sorgente, per la classe *Studente*:

```
public class Studente {
	private String cognome = new String();
	private String nome = new String();
	private String classe = new String();
	private String indirizzo = new String();

	public void cognome(String valoreCognome) {
		this.cognome = valoreCognome;
	}
	public void nome(String valoreNome) {
		this.nome = valoreNome;
	}
	public void classe(String valoreClasse) {
		this.classe = valoreClasse;
	}
	public void indirizzo(String valoreIndirizzo) {
		this.indirizzo = valoreIndirizzo;
	}
}
```

Ricordiamo che una tabella è un insieme di record logici organizzati in righe e colonne. Per tradurre l'ADT tabella in un modello object oriented possiamo impiegare un array monodimensionale (un vettore) di variabili reference di un tipo di classe (record).

Per creare una tabella con i dati degli studenti, possiamo creare un vettore di variabili reference della classe *Studente* con la seguente istruzione:

```
Studente archivioStudenti[] = new Studente[n];
```

dove la variabile intera *n* contiene il numero di studenti da elaborare. Ricordando che un array è un oggetto, l'indirizzo *archivioStudenti* rappresenta "un oggetto di variabili reference". L'istruzione precedente crea in memoria soltanto l'oggetto array, ma non le istanze dei singoli *Studenti*, che prima di essere usate devono essere allocate con comandi del tipo:

archivioStudenti[i] = **new** Studente(*lista argomenti*);

A esempio, nel seguente frammento di codice:

```
public class Studente {
	// Dichiarazione dei membri della classe "Studente".
	...
	public static void main(String args[]) {
		Studente archivioStudenti[] = new Studente[100];	// Array di reference degli oggetti
della classe "Studente".
		archivioStudenti[0] = new Studente();
		archivioStudenti[0].cognome("Giordano");
		archivioStudenti[0].nome("Abramo Gerardo");
		archivioStudenti[0].classe("2C");
		archivioStudenti[0].indirizzo("strada Padana Superiore n.16 - 20096 Pioltello - Italia");
	}	// Fine del metodo "main()".
}	// Fine della classe.
```

si inizializza il primo oggetto (riga) dell'*archivioStudenti* (array di 100 reference) con i dati di uno studente.

Introduzione all'ereditarietà e output di un oggetto

Il Java realizza il meccanismo dell'OOP dell'ereditarietà permettendo, come approfondiremo in seguito, a una classe (denominata derivata o sottoclasse) di ereditare gli attributi e i metodi disponibili in un'altra classe antenata, denominata superclasse o di base. Un metodo ereditato da una classe genitrice può essere impiegato in una sottoclasse senza modifiche oppure il suo corpo può essere ridefinito o sovrapposto (*overriding*) per adattarlo a nuovi comportamenti della classe figlia.

Il meccanismo dell'ereditarietà permette di definire modelli object oriented ad albero di classi (gerarchie di classi) molto complessi alla cui radice si trova sempre, per default, la classe predefinita Object.

Poiché tutte le classi sono sottoclassi di *Object*, i metodi definiti nel suo corpo sono disponibili in qualsiasi classe; inoltre, per lo stesso motivo precedente, *Object* viene raramente rappresentata nei modelli OO delle classi. Un metodo utile di *Object* è quello toString() che può essere ridefinito in una qualsiasi classe per restituire una stringa che rappresenta lo stato interno incapsulato di un oggetto. Lo scheletro generale del metodo *toString()*, da inserire nel corpo di una classe, come nell'esempio seguente:

```
public class Classe {
	private int a = 1;
	private int b = 2;
	private int c = 3;

	public String toString() {
		return ("(" + "a = " + this.a + ", b = " + this.b + ", c = " + this.c + ")");
	}

	public static void main(String args[]) {
		Classe oggettoClasse = new Classe();
		System.out.print("\n\n\n\n\n\t" + oggettoClasse.toString() + "\n\n\n\n\n");
	}
}
```

L'esecuzione del metodo main() precedente produce come output lo stato interno incapsulato delle variabili intere private della classe *Classe*.

(a = 1, b = 2, c = 3)

Oggetti annidati

All'interno del corpo di una classe gli attributi possono appartenere a un'altra classe. Un'istanza di una classe contenuta nello spazio di visibilità di un'altra definisce un oggetto annidato. Seguendo la terminologia dell'OOP, gli oggetti annidati permettono di realizzare, in un programma sorgente, le relazioni di aggregazione (del tipo: un oggetto è *parte di* un altro) individuate durante la fase di progettazione. In generale, per annidare un oggetto di una classe in un'altra si deve:

1. dichiarare una variabile reference della classe annidata come un attributo di istanza privato;
2. allocare un nuovo oggetto annidato nel costruttore della classe esterna mediante l'operatore **new**.

Nella classe Studenti, un indirizzo è, a sua volta, un oggetto/record formato dai campi: viaPiazza, cap, città e nazione. Per modellare questa realtà, possiamo quindi creare una nuova classe *Indirizzo* con il codice seguente:

```
public class Indirizzo {
	private String viaPiazza = new String();	// Attributo privato.
	private String cap = new String();		// Attributo privato.
	private String città = new String();		// Attributo privato.
	private String nazione = new String();		// Attributo privato.

	public void viaPiazza(String valoreViaPiazza) {
		this.viaPiazza = valoreViaPiazza;
	}
	public void cap(String valoreCap) {
		this.cap = valoreCap;
	}
	public void città(String valoreCittà) {
		this.città = valoreCittà;
	}
	public void nazione(String valoreNazione) {
		this.nazione = valoreNazione;
	}
}	// Fine classe.
```

Per annidare un oggetto del tipo Indirizzo nella classe *Studente* dobbiamo modificare il suo corpo nel seguente modo:

```
public class Studente {
	private String cognome = new String();	// Attributo privato.
	private String nome = new String();		// Attributo privato.
	private String classe = new String();	// Attributo privato.
	private Indirizzo indirizzo;		// Variabile reference della classe "Indirizzo".

	public Studente () {			// Costruttore.
		indirizzo = new Indirizzo();		// Allocazione nuovo oggetto della classe
"Indirizzo".
	}
	public void cognome(String valoreCognome) {
```

```
                this.cognome = valoreCognome;
        }
        public void nome(String valoreNome) {
                this.nome = valoreNome;
        }
        public void classe(String valoreClasse) {
                this.classe = valoreClasse;
        }
        public static void main(String args[]) {
                Studente archivioStudenti[] = new Studente[100];      // Array di variabili reference degli
oggetti della classe “Studente”.
                archivioStudenti[0] = new Studente();                 // Creazione del primo oggetto della
classe “Studente” nella prima riga dell'array.
                archivioStudenti[0].cognome("Giordano");
                archivioStudenti[0].nome("Abramo Gerardo");
                archivioStudenti[0].classe("2C");
                archivioStudenti[0].indirizzo.viaPiazza("strada Padana Superiore n.16");
                archivioStudenti[0].indirizzo.cap("20096");
                archivioStudenti[0].indirizzo.città("Pioltello");
                archivioStudenti[0].indirizzo.nazione("Italia");
        }
}
```

L'accesso ai dati dell'oggetto *indirizzo*, annidato in un'istanza del tipo *Studente*, avviene applicando in sequenza l'operatore punto con una forma del tipo:

oggettoStudente.indirizzo.metodoClasseIndirizzo(lista argomenti);

Questa versione del programma, per la gestione dell'*archivioStudenti*, richiede l'uso di due file sorgente, *Indirizzo.java* e *Studente.java*, che contengono il codice delle corrispondenti classi.

Classi interne

Il Java permette di dichiarare una classe all'interno del corpo di un'altra, creando una classe interna (*inner class*). Una classe interna ha accesso a tutti i membri (attributi di istanza e metodi), anche privati, della classe in cui è contenuta; viceversa, la classe esterna non può accedere agli elementi di quella interna. Una classe interna può anche essere inclusa in un'altra senza un nome; in questo caso si denomina classe interna anonima. Le classi interne trovano applicazione nella creazione di gruppi di classi logicamente collegate tra loro e nella gestione degli eventi generati nelle GUI, che analizzeremo nel seguito.

Quando si compila un file sorgente di una classe esterna (*Esterna.java*) che ne contiene una interna (di nome *Interna*), il compilatore crea due file *.class*. Il primo *Esterna.class* contiene il bytecode della classe esterna, mentre il secondo, di nome *Esterna$Interna.class*, memorizza il codice compilato della classe interna.

Per annidare un indirizzo in uno studente, possiamo inserire il corpo della classe *Indirizzo* in quella *Studente*.

```
public class Studente {   // Classe esterna.
        // Dichiarazione dei membri della classe esterna “Studente”.
        ...
        public class Indirizzo {   // Classe annidata interna.
                // Dichiarazione dei membri della classe interna “Indirizzo”.
                ...
        }   // Fine della classe interna “Indirizzo”.

        public static void main(String args[]) {
                Studente archivioStudenti[] = new Studente[100];
                archivioStudenti[0] = new Studente();
                // Inizio elaborazione.
                ...
        }   // Fine del metodo “main()”.
}   // Fine della classe esterna “Studente”.
```

Nell'applicazione precedente, *Indirizzo* è una classe interna annidata in quella esterna *Studente*. La compilazione del file sorgente *Studente.java* produce i file in bytecode *Studente.class* e *Studente$Idirizzo.class*.

Programmi sorgente object oriented multifile

Finora abbiamo sempre utilizzato gli oggetti di una classe richiamandoli all'interno dei metodi *main()* di un'applicazione console, dichiarati nel medesimo file sorgente con la classe stessa. Questa tecnica è ideale nella fase di testing (del tipo *black box*) di una singola classe, ma non è efficiente durante il richiamo di una oppure più classi in elaborazioni complesse. L'approccio alternativo è quello di applicare la tecnica di progettazione modulare per creare un programma finale formato da una serie di moduli.

Modulo object oriented

In Java, un file con la dichiarazione di una classe costituisce un modulo object oriented.
Un file sorgente con la dichiarazione di una classe Java:

- ✓ deve avere lo stesso nome della classe (con estensione *.java*) e costituisce una unità di compilazione autonoma (nel file compilato in bytecode *.class*);
- ✓ realizza, in un programma sorgente, un modulo dei diagrammi omonimi di Booch della fase di progettazione.

Possiamo quindi pensare a un software object oriented complesso come formato da un insieme di moduli, ognuno dei quali realizza una singola classe. Tra i moduli di un progetto, il Java distingue il modulo principale (a sua volta una classe) che è quello che in un'applicazione, console o con GUI, contiene il metodo *main()* da cui si avvia l'elaborazione.
Nel modulo principale, per richiamare oggetti delle altre classi possiamo (in alternativa):

- ✓ includere i file .class nella stessa cartella del file in bytecode con il modulo principale;
- ✓ utilizzare il comando import, se abbiamo realizzato le classi richiamate in un package autonomo.

A scopo esemplificativo, scriviamo un'applicazione console (modulo principale), salvata nel file sorgente *ModuloMain.java*, che richiama oggetti della classe *EquazioneSecondoGrado* e *Veicolo*.

```
public class ModuloMain {
	public static void main(String args[]) {
		Veicolo auto = new Veicolo();
		EquazioneSecondoGrado equazione = new EquazioneSecondoGrado();
		// Elaborazione degli oggetti del tipo sia Veicolo sia EquazioneSecondoGrado.
		...
	}   // Fine del metodo "main()".
}   // Fine della classe "MetodoMain".
```

Prima della compilazione del file *MetodoMain.java*, dobbiamo pero copiare i file con il bytecode delle due classi (*EquazioneSecondoGrado.class* e *Veicolo.class*) nella cartella dell'applicazione console; in caso contrario, non trovando le classi compilate, la traduzione si arresta con degli errori di compilazione del tipo cannot resolve symbol. Terminata la compilazione del file *MetodoMain.java*, la cartella, con la nostra prima applicazione OO multifile, conterrà i file in bytecode del modulo principale (*ModuloMain.class*) e di quelli delle due classi richiamate. L'avvio dell'applicazione multifile. I metodi *main()* delle classi *EquazioneSecondoGrado.class* e *Veicolo.class* non sono più utilizzati perché l'elaborazione si avvia nel metodo *main()* della classe *ModuloMain*, richiamata dall'interprete Java.
I concetti precedenti non sono delle regole sintattiche del linguaggio Java, ma delle linee guida seguite da gruppi di programmatori per costruire software sorgente efficienti, riusabili, portabili e ben documentati. Per la tecnologia Java, vedremo che questa soluzione equivale ad aver progettato un nuovo package di default ovvero una libreria di classi senza nome (*unnamed package*).
In alternativa all'approccio appena descritto, la tecnologia Java permette di creare un unico file sorgente in cui vengono inserite le dichiarazioni di tutte le classi richiamate nel programma più la classe da cui si avvia l'esecuzione. In questo caso, soltanto la classe che avvia l'elaborazione deve essere dichiarata col modificatore **public**, il cui identificatore deve diventare anche il nome del file sorgente *.java*. La compilazione di un file sorgente con più classi produce un insieme di file *.class*, ognuno dei quali contiene il bytecode di una singola classe.
Per collaudare quest'ultimo approccio, modifichiamo il file sorgente *MetodoMain.java* inserendo (con operazioni del tipo "copia e incolla" dell'editor di testi) i codici sorgente delle due classi *Veicolo* ed *EquazioneSecondoGrado*. Eliminiamo quindi il modificatore **public** a queste ultime classi lasciandolo solo per *ModuloMain*, come schematizzato nello scheletro di codice sorgente che segue.

```
public class ModuloMain {
	public static void main(String args[]) {
		Veicolo auto = new Veicolo();
		EquazioneSecondoGrado equazione = new EquazioneSecondoGrado();
		// Elaborazione degli oggetti del tipo sia Veicolo sia EquazioneSecondoGrado.
	}   // Fine del metodo "main()".
}   // Fine della classe "MetodoMain".
class Veicolo {
	// Dichiarazione dei membri della classe "Veicolo".
	...
```

```
}       // Fine della classe "Veicolo".
class EquazioneSecondoGrado {
        // Dichiarazione dei membri della classe "EquazioneSecondoGrado".
        ...
}       // Fine della classe "EquazioneSecondoGrado".
```

La compilazione del precedente file sorgente produce tre file di classe in bytecode: *ModuloMain.class* (modulo principale), *Veicolo.class* e *EquazioneSecondoGrado.class*. L'avvio dell'applicazione avviene richiamando, con l'interprete, il file in bytecode *ModuloMain.class*.
Quest'ultimo tipo di realizzazione di un software è impiegato raramente dai programmatori Java e soltanto nella fase di testing (a livello di integrazione tra classi) per il collaudo della comunicazione tra un numero limitato di classi.

Le interfacce

Finora abbiamo sempre considerato le classi come l'unico tipo di dato creato dai programmatori. La tecnologia Java dispone però anche delle interfacce, per dichiarare tipi di dati utilizzati nelle applicazioni per modellare particolari realtà.
Una interfaccia (*interface*) è un tipo di dato, dichiarato con il costrutto **`interface`** del linguaggio, che deve essere interamente implementato in una oppure più classi differenti.
Non dobbiamo confondere "una interfaccia" (particolare tipo di dato) con "l'interfaccia di una classe" (insieme dei metodi pubblici) che rappresentano concetti molto diversi tra loro nella tecnologia Java. Per dichiarare un tipo interfaccia si impiega la keyword **`interface`** al posto di quella **`class`**. Lo scheletro del codice sorgente per la dichiarazione di una nuova interfaccia, con la keyword **`interface`**, è analogo a quello di una classe (con il costrutto **`class`**), a eccezione delle tre sostanziali differenze elencate nel seguito.

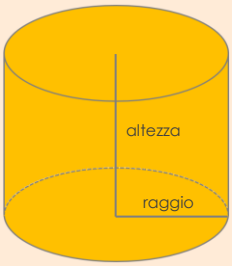
1. Tutti gli attributi di un'interfaccia devono essere delle costanti pubbliche statiche, dichiarati quindi con i modificatori **`final static`** e inizializzati con un valore.
2. Tutti i metodi nel costrutto **`interface`** sono pubblici e devono essere dichiarati con il modificatore **`abstract`**.
3. Non possiede costruttori.

In Java, un metodo astratto (*abstract method*) è un sottoprogramma che non contiene un blocco di istruzioni, per cui non implementa alcuna elaborazione.
Seguendo le indicazioni precedenti, la dichiarazione di una nuova interfaccia appare con uno scheletro di codice sorgente del tipo seguente:

```
[public]optional interface NomeInterfaccia {
        <[public final static tipoDato NOMECOSTANTE = espressione;]optional>repeat
        <[public abstract tipoDatoRestituito nomeMetodo(lista paarametri);]optional>repeat
}
```

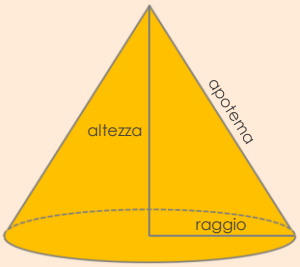
Come per le classi, dopo aver progettato un'interfaccia, il suo codice deve essere scritto in un file sorgente *.java* che viene tradotto, come un'unità di compilazione autonoma, in un file in bytecode *.class*.
Il Java dispone anche delle classi astratte, che possiamo considerare come un tipo di dato intermedio tra un'interfaccia e una classe propriamente detta. Quando studieremo il polimorfismo vedremo che una classe astratta dispone di metodi astratti (non implementati, come le interfacce) e di metodi (con un corpo definito, come le classi propriamente dette).
Come esempio di applicazione delle interfacce consideriamo il seguente problema geometrico.
Realizzare un software, per un'applicazione di grafica assistita dal computer (CAD: Computer Aided Design) a tre dimensioni (3D) per l'elaborazione dei tre solidi di rotazione: cilindro, cono e sfera (ottenuti facendo ruotare, rispettivamente, un rettangolo, un triangolo e un semicerchio intorno a una retta del loro piano). L'utente del CAD 3D ricorda ai programmatori le seguenti relazioni per l'elaborazione dei tre solidi di rotazione.

CILINDRO



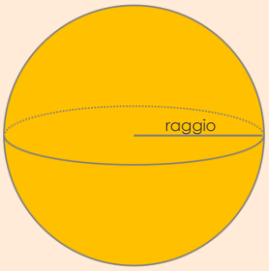
$Superficie = 2 \cdot \pi \cdot raggio \cdot (altezza + raggio)$
$Volume = \pi \cdot (raggio)^2 \cdot altezza$

CONO



$Superficie = \pi \cdot raggio \cdot (apotema + raggio)$
$Volume = \frac{\pi \cdot (raggio)^2 \cdot altezza}{3}$

SFERA



$Superficie = 4 \cdot \pi \cdot raggio^2$
$Volume = \frac{4}{3} \cdot \pi \cdot raggio^3$

I tre solidi di rotazione della figura precedente sono diversi tra loro, eppure hanno una costante (π) e dei comportamenti (*superficie* e *volume*) in comune. Gli algoritmi dei metodi sono però differenti tra loro, perché i calcoli delle superfici e dei volumi sono rappresentati sempre da formule differenti. Per descrivere questa realtà possiamo dichiarare una nuova interfaccia *SolidoRotazione* in cui sono dichiarate la costante PIGRECO e i metodi pubblici *superficie()* e *volume()* comuni e quindi condivisi tra tutti i solidi di rotazione.

```
public interface SolidoRotazione {
        public static final double PIGRECO = 3.14159265;

        public abstract double superficie ();
        public abstract double volume();
}       // Fine interfaccia.
```

Il codice precedente deve essere salvato in un file *SolidoRotazione.java* e deve essere compilato in quello con il bytecode *SolidoRotazione.class*.

Per convenzione del linguaggio, essendo tutti i metodi di un'interfaccia automaticamente pubblici e astratti, il programmatore può evitare di far precedere ai nomi dei sottoprogrammi le parole chiave **public** e **abstract**. In modo analogo anche le costanti pubbliche e statiche possono essere semplicemente dichiarate con il loro tipo di dato senza le parole chiave **public static** e **final**.

Utilizziamo quest'ultimo consiglio per riscrivere la dichiarazione dell'interfaccia *SolidoRotazione*, come un "programmatore Java", nel modo seguente.

```
public interface SolidoRotazione {
        double PIGRECO = 3.14159265;

        double superficie ();
        double volume();
}       // Fine interfaccia.
```

Implementazione di un tipo interfaccia

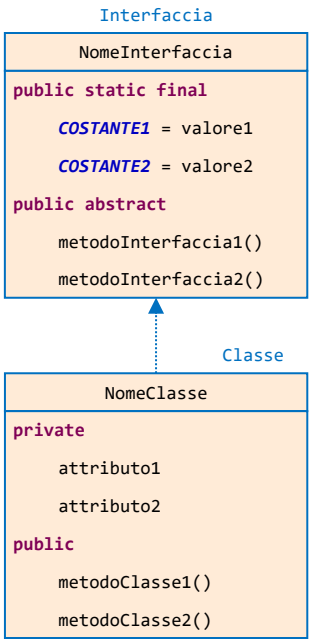
Il compito di un'interfaccia è quello di definire un modello (template), dichiarando costanti e/o comportamenti che dovranno essere implementati obbligatoriamente con il codice sorgente, in altre classi che seguono quel tipo di dato.
Una classe che implementa un'interfaccia:

- ✓ può accedere alle costanti dichiarate nel corpo dell'interfaccia stessa;
- ✓ deve implementare i metodi dell'interfaccia come pubblici, rispettando le firme dichiarate (tipo di dato restituito, identificatore e tipo di dato dei parametri);
- ✓ può eventualmente completare il modello dei dati e il comportamento, aggiungendo propri attributi di istanza e metodi.

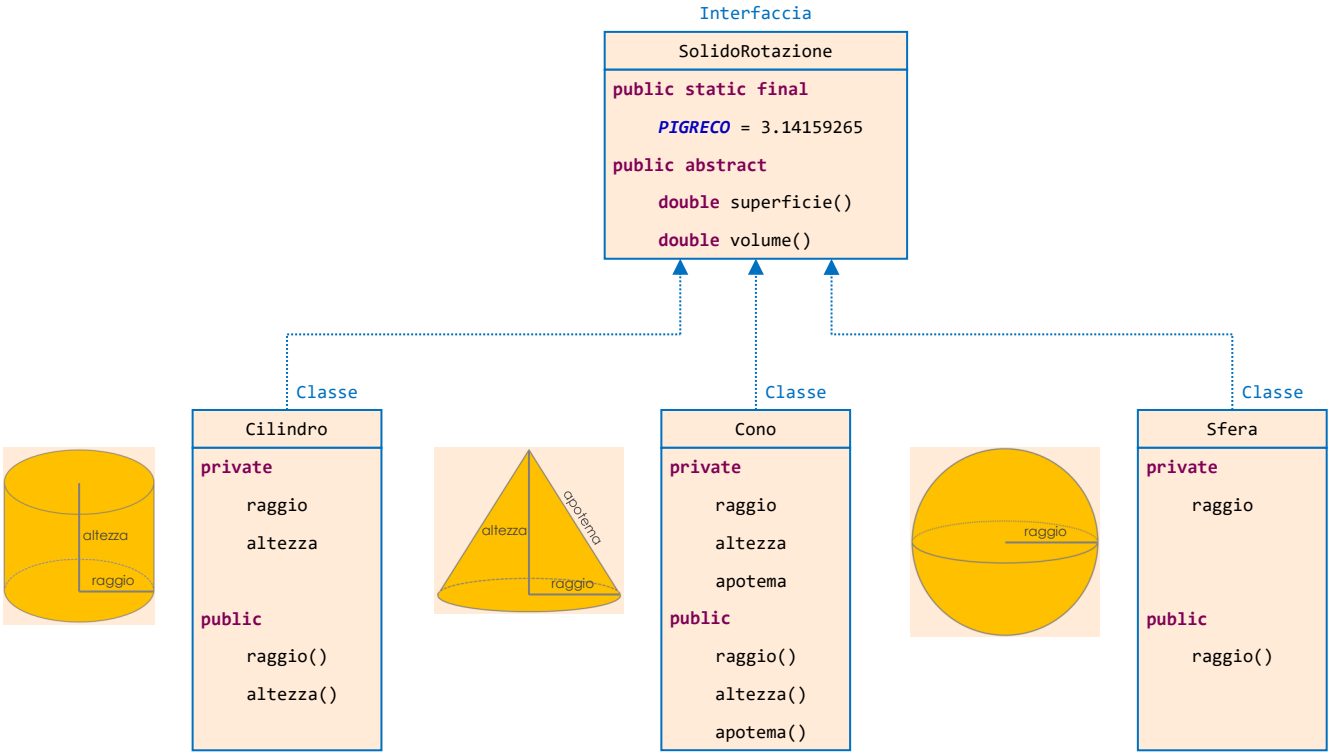
Per dichiarare che una classe implementa una oppure più interfacce, nel suo costrutto **class** si deve aggiungere, dopo il suo identificatore, la parola chiave **implements** seguita dalla lista dei nomi delle interfacce separati da una virgola. Lo scheletro generale di una classe che implementa una o più interfacce si presenta quindi nel seguente modo:

```
[public]optional class NomeClasse implements NomeInterfaccia1 <,nomeInterfaccia2>repeat {
        <public tipoDatoRestituito nomeMetodo(lista parametri) {
                ... // Implementazione del corpo del metodo dell'interfaccia.
        }
        ... // Dichiarazione di attributi propri della classe.
        ... // Dichiarazione di metodi propri della classe.
}       // Fine classe.
```

Nella tecnologia Java, a livello del progetto di un software object oriented, per rappresentare "una classe che implementa un'interfaccia", si impiega un diagramma delle classi di Booch del tipo descritto nella figura che segue.



La freccia ⋯▶ nella figura precedente, orientata da una classe verso un tipo di dato interfaccia, rappresenta in modo grafico il verbo "...implementa...".
I programmatori Java spesso ricorrono alle interfacce per stabilire per stabilire delle similitudini tra classi senza forzare delle relazioni di ereditarietà.
Le similitudini tra i solidi di rotazione cilindro, cono e sfera, possono essere rappresentate mediante il seguente modello object oriented descritto con un diagramma delle classi di Booch.



Sulla base del modello precedente, realizziamo la classe *Cilindro* che implementa l'interfaccia *SolidoRotazione*.

```
public class Cilindro implements SolidoRotazione {
        private double raggio;
        private double altezza;

        public void raggio(double raggio) {this.raggio = raggio;}
        public void altezza(double altezza) {this.altezza = altezza;}
        public double superficie() {return 2.0*PIGRECO*raggio*(altezza+raggio);}
        public double volume() {return PIGRECO*raggio*raggio*altezza;}
}
```

La classe Cilindro:

- ✓ utilizza nel suo corpo la costante PIGRECO;
- ✓ implementa i metodi dichiarati dell'interfaccia *superficie()* e *volume()* (in modo pubblico);
- ✓ aggiunge i propri attributi privati incapsulati *raggio* e *altezza*, realizzando anche i due metodi *raggio()* e *altezza()* per la scrittura del loro valore.

La classe precedente deve essere salvata nel file sorgente *Cilindro.java* e compilata nella stessa cartella che contiene il file *SolidoRotazione.class*, per produrre il bytecode *Cilindro.class*. Le altre classi *Cono* e *Sfera*, che implementano l'interfaccia *SolidoRotazione*, possono essere dichiarate in modo del tutto analogo a quella *Cilindro*, seguendo il modello OO precedente.

Reference, oggetti e interfacce

In un programma:

- ✓ non è possibile creare un oggetto di un tipo interfaccia, anche se è possibile dichiarare una variabile reference;
- ✓ è possibile allocare oggetti di una classe che implementa una oppure più interfacce.

Studieremo in seguito ce l'impiego di variabili reference di un tipo interfaccia è uno dei modi per realizzare in Java il meccanismo del polimorfismo (in fase di esecuzione) dell'OOP.

Dopo aver dichiarato l'interfaccia *SolidoRotazione* e le classi che la implementano, *Cilindro*, *Cono* e *Sfera*, possiamo utilizzare un'applicazione console per effettuare l'elaborazione di questi tipi di figure geometriche.

La seguente applicazione console, basata sulla classe *MainSolidi* (salvata nel file sorgente *MainSolidi.java*), crea un oggetto del tipo *Cilindro* per effettuare elaborazioni su questo tipo di solido di rotazione. Nell'applicazione, per semplicità, assegniamo già dei valori per il raggio e l'altezza (in centimetri).

```
public class MainSolidi {
	public static void main(String args[]) {
		Cilindro cilindro = new Cilindro();
		cilindro.raggio(1.2);
		cilindro.altezza(5.8);
		System.out.print("\n\n\n\n\n\tSuperficie cilindro = " + cilindro.superficie() + " m^2");
		System.out.print("\n\tVolume cilindro     = " + cilindro.volume() + " m^3\n\n\n\n\n");
	}	// Fine metodo "main()".
}	// Fine classe.
```

Per il corretto funzionamento del programma, il file in bytecode *MainSolidi.class* deve essere inserito nella stessa cartella con quelli dell'interfaccia (*SolidoRotazione.class*) e delle classi richiamate (*Cilindro.class*, *Cono.class* e *Sfera.class*).

L'esecuzione dell'applicazione console produce come output il risultato di seguito riportato:

```
Superficie cilindro = 52.77875652 m^2
Volume cilindro     = 26.2385818128 m^3
```

Applicazioni delle interfacce

Nelle applicazioni, le interfacce possono essere impiegate principalmente per scopi descritti nel seguito.

1. Stabilire similitudini tra tipi di dato differenti (l'interfaccia e le classi che la implementano) senza ricorrere ad altri meccanismi dell'OOP, tra cui, in particolare, quello dell'ereditarietà.
2. Dichiarare metodi che altri programmatori devono necessariamente implementare nelle loro classi.
3. Definire un insieme di costanti di utilità generale, che altri programmatori possono utilizzare nei loro programmi.
4. Creare degli oggetti che possono appartenere a più tipi di dato, mediante la dichiarazione di una classe che implementa più interfacce.

Schematizzando, un'interfaccia definisce una sorta di "contratto" firmato tra tipi dato, che altre classi dovranno rispettare al momento della loro implementazione. Una interfaccia è spesso la cima (radice) di un modello gerarchico formato da un albero di classi, dovendo prevedere un comportamento generale dichiarato nel codice sorgente delle classi che la implementano.

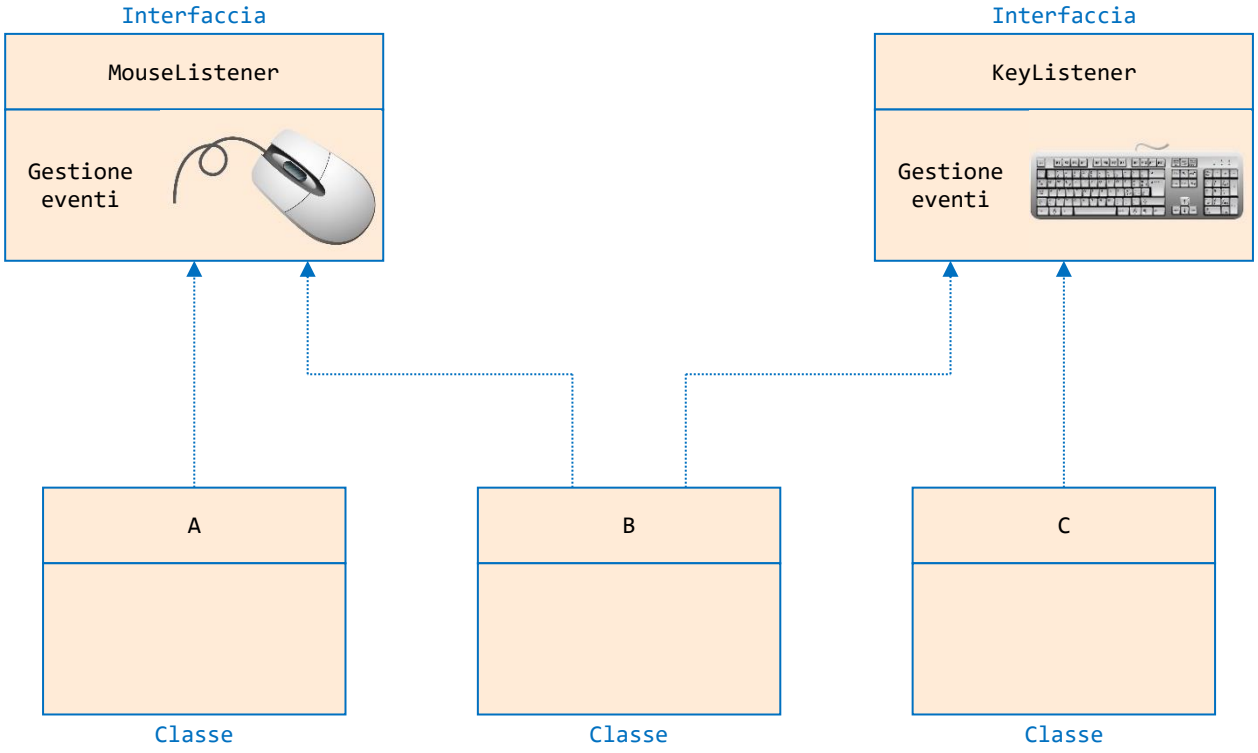
Quest'ultima caratteristica è spesso ricordata tra i programmatori con l'espressione: "*un'unica interfaccia molte implementazioni*".

La documentazione di un'interfaccia, essendo un particolare tipo di classe, avviene elencando le sue costanti e/o i suoi metodi (tutti pubblici) e distribuendo il file in bytecode *.class*.

Implementazione di interfacce multiple

Una classe può impiegare più interfacce, per poter creare nei programmi oggetti che appartengono a più tipi di dato. Nel corpo della classe, il programmatore dovrà obbligatoriamente realizzare tutti i metodi (come pubblici) dichiarati nelle interfacce implementate.

Studieremo in seguito come il Java, per gestire gli eventi generati dal mouse e dalla tastiera, impieghi due interfacce autonome denominate rispettivamente, *MouseListener* e *KeyListener*. Si possono quindi verificare diversi casi, descritti nel modello OO esemplificativo seguente, per cui la classe *A*, che deve gestire solo gli eventi del mouse, implementa solo l'interfaccia *MouseListener*, la classe *C*, che gestisce solo la tastiera, completa il tipo *KeyListener*, mentre la classe *B*, che necessita di entrambe le categorie di eventi, deve implementare tutti i metodi delle due interfacce.



Confronto tra i tipi di dato interfaccia e classe

La tabella seguente riassume le regole generali per la dichiarazione di un tipo di dato interfaccia ponendole a confronto con quelle di un tipo classe.

	Interfaccia	Classe
Sintassi generale	`[public]optional interface NomeInterfaccia { ... // Corpo dell'interfaccia. }`	`[public]optional class NomeClasse { ... // Corpo della classe. }`
Attributi	Possono essere esclusivamente costanti pubbliche statiche (`public static final`).	Non ci sono restrizioni.
Metodi	Possono essere esclusivamente metodi pubblici il cui corpo non è implementato (metodi pubblici astratti: `public abstract`).	Non ci sono restrizioni.
Costruttori	Non possono essere dichiarati	Non ci sono restrizioni.
Dichiarazione variabili reference	Ammesso	Ammesso.
Creazione di un oggetto	Non è ammesso.	Ammesso.

Programmazione della GUI

L'interfaccia grafica utente (GUI) delle applicazioni Java è fondata su classi appartenenti al package Swing (*javax.swing*), che in Java 2 estende e integra le funzioni della libreria AWT (*Abstract Window Toolkit*), disponibile fin dalla versione 1.0 del linguaggio.
Di fronte a una nuova classe della GUI, il programmatore deve studiare:

- ✓ i suoi attributi, che in generale definiscono l'aspetto grafico degli stessi oggetti;
- ✓ i metodi, che ne determinano il comportamento.

Gli attributi sono spesso incapsulati (privati) nelle classi grafiche, per cui la loro scrittura e/o lettura avviene con metodi pubblici di accesso, ad esempio del tipo *scriviAttributo()* e/o *leggiAttributo()*. Per un programmatore, la gestione di un oggetto di una classe grafica si riduce quindi al richiamo di soli metodi, per l'accesso agli attributi privati e alle elaborazioni incapsulate. A differenza di altri linguaggi, il Java non elabora gli eventi degli oggetti della GUI come metodi, ma li gestisce in modo separato mediante opportune classi e interfacce.

Finestre

Una finestra è un oggetto che realizza la comunicazione tra l'utente e il codice del programma.
Una finestra è spesso una istanza complessa formata da un insieme di altri oggetti annidati tra cui i principali sono i pannelli, i quali contengono a loro volta i componenti (*component*) come, a esempio, le etichette, i campi di testo, e i pulsanti di comando. Le finestre e i pannelli, potendo annidare altri oggetti, sono esempi di contenitori (container).
Nel seguito useremo i termini "oggetto grafico" per indicare, in generale, una finestra oppure uno dei suoi elementi annidati.

Gestione degli eventi

Un evento è una qualsiasi azione comunicata al nucleo del sistema operativo. Gli eventi possono essere hardware, quando sono causati da dispositivi periferici del computer (mouse, tastiera ecc.) oppure software, quando sono generati dalle stesse applicazioni. Gli eventi raccolti dal nucleo del sistema operativo sono "recapitati" a tutte le applicazioni in esecuzione (che possono essere più di una se il software di base è multitasking). Il meccanismo di gestione degli eventi del Java ha subìto delle modifiche e, a partire dalla versione 1.1, è basato su un modello denominato a delega degli eventi (*delegation event model*). L'idea alla base del modello è quella di non far gestire gli eventi agli oggetti che li generano (*source object*), ma di "delegare" l'elaborazione dei metodi associati agli eventi a degli oggetti ascoltatori (*listener object*).

Eventi

Un evento (event) del linguaggio è un oggetto che può appartenere a un insieme di classi, ognuna delle quali è specializzata per gestire un insieme di specifiche azioni.
La tabella seguente descrive alcuni metodi delle principali classi che modellano eventi nel Java.

Classe di evento	Descrizione
WindowEvent	Modella gli eventi associati alle finestre. Il metodo *getWindow()* restituisce il nome dell'oggetto che ha generato l'evento.
MouseEvent	Descrive gli eventi generati dal mouse. I metodi *getX()* e *getY()* restituiscono, rispettivamente, le coordinate *X* e *Y* del puntatore del mouse del punto in cui si è verificato l'evento, mentre *getClickCount()* fornisce il numero di clic fatti dall'utente con il pulsante del mouse.
KeyEvent	Descrive gli eventi generati dalla tastiera. La classe definisce numerose costanti (intere) che rappresentano i principali tasti della tastiera quali: VK_SHIFT (tasto SHIFT), VK_CONTROL (CTRL), VK_CANCEL (CANC), VK_ESCAPE (ESC), VK_LEFT (←), VK_UP (↑), VK_RIGHT (→), VK_DOWN (↓), VK_A ... VK_Z (lettere dalla A alla Z), VK_F1 ... VK_F24 (tasti funzione da F1 a F24), VK_UNDEFINED (tasto indefinito) ecc. Il metodo *getKeyChar()* restituisce il carattere del tasto premuto, mentre *getKeyCode()* fornisce un valore intero che corrisponde alle costanti VK_*TASTO*.
ItemEvent	Modella gli eventi generati dalla selezione di un oggetto quali un pulsante di opzione oppure una scelta in un menu. I metodi principali della classe sono: *getItemSelectable()*, che fornisce un reference all'oggetto che ha provocato l'evento; *getItem()*, che restituisce un riferimento alla voce scelta; *getStateChange()*, che restituisce la costante SELECTED, se la voce è stata selezionata, oppure DESELECTED, se l'opzione ha perso la selezione.
ComponentEvent	Modella eventi associati a componenti. Il metodo *getComponent()* restituisce il nome dell'oggetto che ha creato l'evento.

AdjustmentEvent	Descrive eventi generati da un oggetto grafico che può modificare il suo stato (e quindi il suo valore), tipicamente una barra di scorrimento. Il metodo *getAdjustable()* restituisce il nome dell'oggetto che ha provocato l'evento, mentre *getValue()* fornisce il valore attuale associato allo stato dell'oggetto.
ActionEvent	Descrive eventi generici, generati nella GUI. Le costanti (intere) ALT_MASK, CTRL_MASK e SHIFT_MASK permettono di verificare se, durante la creazione dell'evento, l'utente aveva premuto, rispettivamente, i tasti ALT, CTRL e SHIFT. Il metodo *getActionCommand()* restituisce una stringa con il comando che ha causato l'evento.

Ascoltatori di eventi

In Java, una classe, se deve gestire degli eventi deve implementare una oppure più interfacce per l'ascolto di eventi (*listener interface*), contenute nel package *java.awt.event*. Una istanza di una classe che implementa tali interfacce (*listener class*) diventa un oggetto ascoltatore (*listener object*) di eventi.
Una interfaccia listener raccoglie tutti i metodi che descrivono le azioni provocate da una determinata categoria di eventi. I programmatori possono riconoscere queste interfacce dai loro identificatori, che sono del tipo *TipoEventi*Listener. I metodi dichiarati in queste interfacce restituiscono (come parametro passato per reference) un oggetto di una classe di evento che fornisce tutte le informazioni sull'evento stesso. La tabella seguente schematizza le principali categorie di eventi del Java con le corrispondenti interfacce listener e alcuni metodi per la loro gestione.

Come fare per gestire gli eventi ...	Interfacce per l'ascolto di eventi e intestazioni (firme) dei metodi dichiarati nel loro corpo
... associati alla vita di una finestra (apertura, spostamento, riduzione a icona e chiusura)	Interfaccia **WindowListener** ✓ `public void windowActivated(WindowEvent oggettoEvento)`: metodo richiamato quando la finestra diventa attiva. ✓ `public void windowClosing(WindowEvent oggettoEvento)`: metodo richiamato quando l'utente chiude la finestra dal menu di sistema. ✓ `public void windowDeactivated(WindowEvent oggettoEvento)`: metodo richiamato quando la finestra è disattivata (passa in secondo piano perdendo il focus). ✓ `public void windowOpened(WindowEvent oggettoEvento)`: metodo richiamato quando la finestra viene aperta per la prima volta.
... generati dal mouse quali, a esempio, clic e doppio clic.	Interfaccia **MouseListener** ✓ `public void mouseCliced(MouseEvent oggettoEvento)`: metodo richiamato quando l'utente fa clic (preme e rilascia il pulsante del mouse). ✓ `public void mouseEntered(MouseEvent oggettoEvento)`: metodo richiamato quando il mouse entra nella zona occupata da un oggetto grafico. ✓ `public void mouseExited(MouseEvent oggettoEvento)`: metodo richiamato quando il mouse esce dalla zona occupata da un oggetto grafico. ✓ `public void mousePressed(MouseEvent oggettoEvento)`: metodo richiamato quando il pulsante del mouse è premuto (ma non rlasciato). ✓ `public void mouseReleased(MouseEvent oggettoEvento)`: metodo richiamato quando il pulsante del mouse è rilasciato.
... prodotti dal movimento del mouse su una finestra.	Interfaccia **MouseMotionListener** ✓ `public void mouseDragged(MouseEvent oggettoEvento)`: metodo richiamato quando il pulsante di un mouse è premuto e trascinato nell'area di un componente. ✓ `public void mouseMoved(MouseEvent oggettoEvento)`: metodo richiamato quando si muove il puntatore del mouse.
... generati dalla tastiera quando un utente preme un tasto.	Interfaccia **KeyListener** ✓ `public void keyPressed(KeyEvent oggettoEvento)`: metodo richiamato quando un tasto è stato premuto. ✓ `public void keyReleased(KeyEvent oggettoEvento)`: metodo richiamato quando un tasto è stato rilasciato (dopo essere stato premuto). ✓ `public void keyTyped(KeyEvent oggettoEvento)`: metodo richiamato quando si è premuto e rilasciato un tasto (digitazione di un carattere della tastiera).
... generati quando un oggetto grafico e nascosto, spostato, ridimensionato oppure diventa visibile.	Interfaccia **ComponentListener** ✓ `public void componentMoved(ComponentEvent oggettoEvento)`: metodo richiamato quando la posizione di un oggetto grafico si modifica. ✓ `public void componentResized(ComponentEvent oggettoEvento)`: metodo richiamato quando la dimensione di un oggetto grafico cambia.
... generati quando il valore di un testo subisce una modifica.	Interfaccia **TextListener** ✓ `public void textValueChanged(TextEvent oggettoEvento)`: metodo richiamato quando un testo subisce una modifica.

… dovuti alla selezione di un oggetto (a esempio, un pulsante di opzione o una voce in un menu).	Interfaccia **ItemListener** ✓ `public void itemStateChanged(ItemEvent oggettoEvento)`: metodo richiamato quando l'utente sceglie un'opzione (a esempio in un menu o in una casella di selezione).
… generati quando un oggetto grafico diventa quello attivo (riceve il focus) oppure perde il focus.	Interfaccia **FocusListener** ✓ `public void focusGained(FocusEvent oggettoEvento)`: metodo richiamato quando un oggetto riceve il focus (provocato da un'azione della tastiera). ✓ `public void focusLost(FocusEvent oggettoEvento)`: metodo richiamato quando un oggetto perde il focus (provocato da un'azione della tastiera).
… provocati dalla modifica dello stato di un oggetto grafico.	Interfaccia **AdjustmentListener** ✓ `public void adjustmentValueChanged(AdjustmentEvent oggetto)`: metodo richiamato quando un valore di un oggetto grafico è stato modificato.
… generali, prodotti da un'azione di un utente su un oggetto grafico.	Interfaccia **ActionListener** ✓ `public void actionPerformed(ActionEvent oggettoEvento)`: metodo richiamato quando si verifica un qualsiasi generico evento.

Il seguente codice sorgente:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class AscoltaMouse extends JApplet implements MouseMotionListener {

        public void mouseDragged(MouseEvent oggettoEventoMouse) {}
        public void mouseMoved(MouseEvent oggettoEventoMouse) {}
}
```

dichiara che l'applet *AscoltaMouse* è una classe delegata (dal programmatore) ad ascoltare, e quindi a gestire, gli eventi generati dal movimento del mouse. Nel corpo della classe si dovranno quindi implementare tutti i metodi dichiarati nell'interfaccia *MouseMotionListener*.

Se non è necessario gestire un evento in una classe ascoltatore, i programmatori Java non ne implementano il metodo associato (scrivendo il corpo senza istruzioni nel seguente modo: {})

Sorgenti di eventi

Un oggetto sorgente (*object source*) di eventi è un'istanza grafica "autorizzata" dal programmatore a generare eventi, che saranno "raccolti" ed elaborati da un oggetto ascoltatore.
Per default, un oggetto grafico non può generare eventi. Il programmatore deve quindi sempre scegliere quali oggetti possono generare eventi, la cui gestione è delegata a metodi degli oggetti delle classi ascoltatore. Per delegare la gestione degli eventi generati da un oggetto sorgente a uno ascoltatore, si impiegano i metodi di registrazione la cui sintassi generale si presenta con la forma seguente:

oggettoSorgente.add*TipoEventi*Listener(*oggettoAscoltatore*);

Se l'oggetto ascoltatore e/o quello sorgente sono dichiarati nella stessa classe, rispettivamente listener o source, si impiega il reference predefinito **this**.
Nell'esempio precedente, vogliamo gestire gli eventi del movimento del mouse generati nell'applicazione *AscoltaMouse* con i metodi inseriti nel corpo della stessa classe. In questo caso, quindi, gli oggetti sorgente e ascoltatore sono la stessa applicazione (oggetto **this**). Per delegare la gestione degli eventi generati nell'applicazione ai metodi inseriti nella stessa classe, dobbiamo quindi inserire nel metodo *main()* il seguente comando:

```
this.addMouseMotionListener(this);
```

Nel corpo del metodo *mouseMoved()*, al fine di monitorare il movimento del mouse (esclusivamente nella finestra dell'Applet) è possibile richiamare il metodo *showStatus*(stringa), che visualizza ina stringa nella barra di stato del browser richiamando le coordinate attuali del mouse fornite dai metodi *getX()* e *getY()* dell'oggetto del tipo *MouseEvent*.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class AscoltaMouse extends JApplet implements MouseMotionListener {
	public void init() {this.addMouseMotionListener(this);}

	public void mouseDragged(MouseEvent oggettoEventoMouse) {}
	public void mouseMoved(MouseEvent oggettoEventoMouse) {
		String messaggio;
		messaggio = "Posizione del mouse : ( " + oggettoEventoMouse.getX() + ", " +
oggettoEventoMouse.getY() + " )";
		showStatus(messaggio);
	}
}
```

Schematizzando, per elaborare gli eventi in un oggetto grafico il programmatore deve completare i seguenti passi operativi.

1. Dichiarare una oppure più classi ascoltatore che implementano le interfacce *TipoEventiListener* e istanziare gli oggetti ascoltatore.
2. Delegare la gestione degli eventi generati dagli oggetti sorgente a quelli ascoltatore impiegando, per questo, i metodi di registrazione:
   oggettoSorgente.add*TipoEventi*Listener(*oggettoAscoltatore*)
3. Implementare tutti i metodi associati agli eventi delle classi ascoltatore.

Gestione degli eventi con le classi adattatore e interne

Quando un programmatore implementa una classe ascoltatore deve scrivere il codice sorgente di tutti i metodi delle interfacce listener. Per evitare questo lavoro aggiuntivo, il Java associa a ogni interfaccia del tipo *TipoEventi*Listener, che dispone di più metodi, una classe adattatore con un nome *tipoEventi*Adapter, che implementa tutti i metodi dell'interfaccia senza inserire nei loro corpi alcuna istruzione. Se una classe vuole gestire una solo categoria di eventi, può quindi estendere (ereditare) un tipo adapter con una dichiarazione del tipo seguente:

```
public class NomeClasseAscoltatore extends TipoEventiAdapter
```

Per una regola del Java, una classe può soltanto ereditare da un'unica superclasse (ma implementare più interfacce), per cui l'impiego delle classi adattatore permette di implementare soltanto i metodi di una categoria di eventi specifici.

I programmatori Java, per gestire in modo semplice gli eventi, inseriscono delle classi ascoltatore interne direttamente all'interno del corpo della classe dell'applicazione con GUI o dell'applet. Con questo accorgimento lo scheletro generale del codice sorgente di un'applicazione con GUI che deve gestire degli eventi diventa il seguente:

```
public class NomeClasseSorgente extends JFrame |or JApplet {
        oggetto1Sorgente.addTipoEventiListener(new NomeClasseAscoltatore()); // Delega eventi.
        oggetto2Sorgente.addTipoEventiListener(new NomeClasseAscoltatore()); // Delega eventi.
        ...
        class NomeClasseAscoltatore implements TipoEventiListener |or extends TipoEventiAdapter {
                ... // Implementazione metodi delle interfacce listener oppure della classe Adapter.
        } // Fine classe ascoltatore interna.
} // Fine classe esterna.
```

Le classi interne ascoltatore possono poi essere più di una, sulla base del tipo di eventi (del mouse, della tastiera ecc.) che si devono elaborare. Come possiamo osservare nello schema di codice precedente, utilizzando una classe ascoltatore interna un oggetto listener può essere creato direttamente nell'argomento della chiamata di un metodo di registrazione con una forma del tipo `new NomeClasseAscoltatore()`.

Supponiamo di voler sviluppare lo scheletro generale di un'applicazione con GUI (basata sulla classe *FinestraGUI*) in grado di visualizzare il messaggio "Grazie per aver utilizzato questo programma.", prima della chiusura della finestra del programma stesso. Per questo obiettivo possiamo impiegare il seguente codice sorgente, che impiega gli eventi associati alle finestre (*window*).

```
import javax.swing.*;
import java.awt.event.*;

public class FinestraGUI extends JFrame {
        public FinestraGUI() {
                this.setTitle("Gestione eventi finestra");
                this.setBounds(0, 0, 1900, 1000);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setVisible(true);
                this.addWindowListener(new Ascoltatore());
        }
        public static void main(String args[]) {
                FinestraGUI finestra = new FinestraGUI();
        }
        class Ascoltatore extends WindowAdapter {
                public void windowClosing(WindowEvent oggettoEventoFinestra) {
                        String messaggio = "Grazie per aver utilizzato questo programma.";
                        JOptionPane.showMessageDialog(null, messaggio);
                        System.exit(0);
                }
        }
}
```

Nel corpo dell'evento si è richiamato il metodo statico *System.exit(0)* della classe System, che interrompe l'esecuzione della JVM in modo corretto se l'argomento intero d'ingresso vale zero.

I pannelli

Nel sistema grafico del Java, un componente (a esempio un pulsante o un campo di testo) non può essere aggiunto direttamente a una finestra *JFrame* o a un'area grafica *JApplet*.
Il modo più semplice per aggiungere dei componenti in una finestra è quello di inserirli in un pannello (*panel*), ovvero un oggetto della classe contenitore *JPanel*.
Per inserire nuovi componenti in una finestra *JFrame* oppure *JApplet*, il programmatore deve quindi completare i seguenti tre passi operativi.

1. Creare un nuovo oggetto della classe *JPanel* con un comando del tipo:
   JPanel *oggettoPannello* = new JPanel();
2. Creare nuovi oggetti componenti e aggiungerli al pannello, impiegando il metodo *add()* con la seguente sintassi:
   oggettoPannello.add(*oggettoComponente*);
3. Impostare il pannello all'interno di una finestra, richiamando il metodo *setContentPane()* delle classi *JFrame* e *JApplet*:
   oggettoFinestra.setContentPane(*oggettoPannello*);

Il seguente frammento di codice sorgente:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class FinestraGUI2 extends JFrame {
	public FinestraGUI2() {
		this.setTitle("FinestraGUI2 con pannello e bottoni.");
		this.setBounds(0, 0, 1900, 1000);
		this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

		JPanel pannello = new JPanel();
		JButton pulsante1 = new JButton("PULSANTE_1");
		JButton pulsante2 = new JButton("PULSANTE_2");
		pannello.add(pulsante1);
		pannello.add(pulsante2);
		this.setContentPane(pannello);
		this.setVisible(true);

		this.addWindowListener(new AscoltatoreFinestra());
		pulsante1.addMouseListener(new AscoltatorePulsante1());
		pulsante2.addMouseListener(new AscoltatorePulsante2());
	}
	public static void main(String args[]) {
		FinestraGUI2 finestra = new FinestraGUI2();
	}
	class AscoltatorePulsante1 extends MouseAdapter {
		public void mouseClicked(MouseEvent oggettoEventoMouse) {
			String messaggio = "Hai premuto il PULSANTE_1.";
			JOptionPane.showMessageDialog(null, messaggio);
		}
	}
	class AscoltatorePulsante2 extends MouseAdapter {
		public void mouseClicked(MouseEvent oggettoEventoMouse) {
			String messaggio = "Hai premuto il PULSANTE_2.";
			JOptionPane.showMessageDialog(null, messaggio);
		}
	}
}
```

Aggiunge due nuovi componenti pulsanti (oggetti della classe *JButton*) sia a un'applicazione con GUI sia a un'applet.

La disposizione degli oggetti grafici in una finestra contenitore determina il layout ovvero la disposizione reciproca dei componenti all'interno del pannello. La gestione dell'aspetto di un pannello è affidata a della classi speciali denominate appunto gestori di layout (*layout manager*). Se non si specifica alcun gestore di layout, il java applica quello di default, che prevede di aggiungere i componenti da sinistra a destra a partire dall'alto. Se arrivati al termine di una riga del pannello, l'area per visualizzare un componente non è sufficiente, il gestore di layout visualizza l'oggetto nella riga successiva, e così fino a esaurire lo spazio verticale del contenitore.

Componenti

Nel seguito descriveremo i principali componenti disponibili in Java nella libreria Swing, mettendo in evidenza i loro attributi incapsulati (controllati da metodi di accesso), i metodi e gli eventi più significativi generati.

Icone

Una icona (*icon*) è un'immagine grafica (memorizzata in un file *.gif*, *.jpg* ecc.), in genere assegnata ad altri componenti. In Java, una icona è un oggetto della classe ImageIcon con un'istruzione del tipo seguente:

ImageIcon *oggettoIcona* = **new** ImageIcon("*URL_FileImmagine*");

Nel costruttore si deve precisare l'URL del file che contiene l'immagine. Incontreremo nel seguito esempi di utilizzo di questo tipo di oggetto grafico.

Etichette

Una etichetta (*label*) è il componente principale per visualizzare il risultato di un'elaborazione, sotto forma di un testo che non può essere modificato dall'utente, oppure creare didascalie per altri componenti.
In Java un'etichetta è un oggetto della classe JLabel creato con un'istruzione del tipo seguente:

JLabel *oggettoEtichette* = **new** JLabel(*testoEtichetta* [, allineamento]$_{optional}$);

Il costruttore di un'etichetta la inizializza con un testo ed eventualmente la allinea usando come allineamento le costanti statiche *SwingConstants.LEFT* oppure *SwingConstants.RIGHT* o infine *SwingConstants.CENTER*. Il principale attributo incapsulato di un'etichetta è il suo testo, che può essere modificato con il metodo *setText(testo)*. A un'etichetta può essere anche assegnata un'icona richiamando il metodo *setIcon(oggettoIcona)*.
Tutti i componenti dispongono dei metodi *setEnabled(valoreLogico)* e *setVisible(valoreLogico)* (ereditati dalla classe *JComponent*), che permettono, rispettivamente, di abilitare/disabilitare oppure rendere visibile/invisibile un oggetto, impostando i valori dell'argomento *valoreLogico* a **true**/**false**.
L'esecuzione del seguente costruttore, inserito in una finestra *JFrame* (classe *FinestraGUI3*), mostra in output l'immagine icona di un pasticcino.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class FinestraGUI3 extends JFrame {
	boolean visualizzaImmagine = false;
	public FinestraGUI3 () {
		this.setTitle("ICONA PASTICCINO");
		this.setBounds(0, 0, 1900, 1000);
		this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		JPanel pannello = new JPanel();
		ImageIcon iconaPasticcino = new ImageIcon("Pasticcino.jpg");
		JLabel notaPasticcino = new JLabel("Icona pasticcino");
		notaPasticcino.setIcon(iconaPasticcino);
		pannello.add(notaPasticcino);
		this.setContentPane(pannello);
		this.setVisible(true);
	}
	public static void main(String args[]) {
		FinestraGUI3 finestra = new FinestraGUI3();
	}
}
```

Campi di testo

Un campo di testo (*text field*) è il componente principale per effettuare l'input dei dati (con la tastiera) in un'applicazione. In Java, un campo (o spesso casella) di testo è un oggetto della classe JTextField che può essere creato con un'istruzione della forma seguente:

JTextField *oggettoCampoTesto* = **new** JTextField([*testoIniziale*]$_{optional}$, [*numeroCaratteri*]$_{optional}$);

Il costruttore permette di creare un campo di testo vuoto con una certa lunghezza in caratteri (JTextField(*numeroCaratteri*)) è un testo iniziale (JTextField(*testoIniziale*)) oppure specificare entrambi (testo e lunghezza in caratteri). Il principale attributo incapsulato di questo componente è il testo contenuto, che può essere gestito con i metodi schematizzati nella tabella che segue.

Come fare per ...	Si impiega il metodo ...
... leggere il contenuto di un campo di testo.	getText()
... modificare il contenuto di un campo di testo.	setText(testo)
... vietare la modifica del contenuto di un campo di testo.	setEditable(scelta) con l'argomento scelta uguale a **false**

Per creare un campo per l'immissione di una password, il Java dispone di una classe specializzata *JPasswordField*, il cui unico carattere visualizzato agli utenti si imposta con il metodo *setEchoChar(carattere)*. I principali eventi generati dai campi di testo sono quelli associati alla tastiera (interfaccia ascoltatore *KeyListener*) e legati alla modifica del testo (*TextListener*):

Il seguente metodo *init()*, inserito in un'applet, definisce due campi di testo per l'inserimento di un nome utente e una password.

```
import java.awt.*;
import javax.swing.*;

public class Login extends JApplet {
	public void init() {
		JPanel pannello = new JPanel();
		JLabel etichettaNomeUtente = new JLabel("Nome utente:");
		JLabel etichettaPassword = new JLabel("Password:");
		JTextField campoNomeUtente = new JTextField(15);
		JPasswordField campoPassword = new JPasswordField(15);
		campoPassword.setEchoChar('*');
		pannello.add(etichettaNomeUtente);
		pannello.add(campoNomeUtente);
		pannello.add(etichettaPassword);
		pannello.add(campoPassword);
		this.setContentPane(pannello);
	}
}
```

Campi di testo multilinea

Il Java dispone di un campo di testo multilinea denominato area di testo (*text area*), la cui classe JTextArea dispone di alcuni tra i comportamenti tipici di un'applicazione di videoscrittura. Il costruttore della classe *JTextArea*, usato come nella seguente istruzione, permette di costruire un'area di testo, specificando un eventuale testo iniziale e il numero di righe e di colonne occupato.

JTextArea *oggettoAreaTesto* = **new** JTextArea([*testo*,]$_{optional}$ *righe*, *colonne*);

Il principale attributo incapsulato è il testo racchiuso nel componente, che può essere gestito con i seguenti metodi.

Come fare per …	Si impiega il metodo …
… scrivere/leggere il contenuto dell'intera area di testo.	setText(*testo*) / getText()
… leggere la parte del testo selezionata dall'utente.	getSelectedArea()
… aggiungere un testo al termine di quello già presente.	append(*testoDaInserire*)
… andare a capo automaticamente al termine di una linea.	setLineWrap(**true**)

I principali eventi generati dalle aree di testo sono quelli dei campi di testo.

Il seguente frammento di codice sorgente, inserito nel costruttore di una finestra con GUI o nel metodo *init()* di un'applet, produce un risultato del tipo presentato nella figura che segue, mentre l'utente sta digitando una frase.

```
import javax.swing.*;

public class AreaDiTesto extends JFrame {
	public AreaDiTesto() {
		this.setTitle("Area di testo multilinea");
		this.setBounds(0, 0, 500, 200);
		this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		this.setVisible(true);
		JPanel pannello = new JPanel();
		JLabel labelAreaTesto = new JLabel("Area di testo");
		JTextArea areaDiTesto = new JTextArea(5, 15);
		areaDiTesto.setLineWrap(true);
		pannello.add(labelAreaTesto);
		pannello.add(areaDiTesto);
		this.setContentPane(pannello);
	}
	public static void main(String args[]) {
		AreaDiTesto applicazione = new AreaDiTesto();
	}
}
```

Pulsanti

Un pulsante o bottone (*button*) è il componente privilegiato per far avviare un'elaborazione in una finestra di un'applicazione o di un'applet. In Java, i pulsanti sono oggetti della classe JButton e possono essere creati mediante un'istruzione della seguente forma:

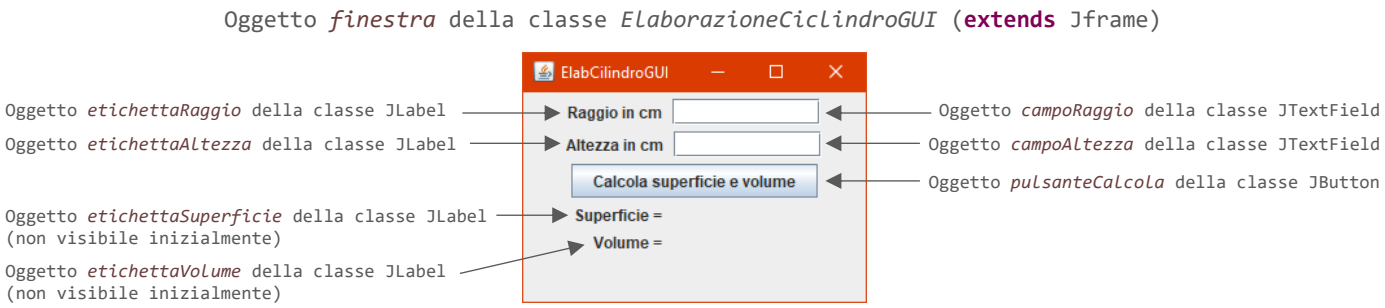
JButton *oggettoPulsante* = **new** JButton([*testo*]$_{optional}$, [*oggettoIcona*]$_{optional}$);

Il costruttore permette di visualizzare nel bottone un testo e/o una immagine definita da un oggetto icona. Il principale attributo di un bottone è il suo testo, che può essere modificato mediante il metodo *setText(testo)*. A un bottone può essere anche assegnata un'icona richiamando il metodo *setIcon(oggettoIcona)*.

L'evento principale generato da un pulsante è quello clic del mouse (metodo *mouseClicked()*), gestito mediante una classe ascoltatore che implementa l'interfaccia *MouseListener*.

Come esempio di applicazione dei componenti fin qui studiati, realizziamo un'applicazione con GUI per il calcolo della superficie e del volume di un cilindro. In generale, dopo aver risolto il problema con una oppure più classi, il compito dell'interfaccia grafica è quello di:

1. Impostare gli attributi incapsulati negli oggetti delle classi, leggendo i valori, a esempio scritti dall'utente in campi di testo;
2. Richiamando i metodi che forniscono i risultati dell'elaborazione (incapsulati nelle classi), ad esempio mediante il metodo generato dall'evento clic dell'operatore su un bottone.
3. Abbiamo già visto che il problema del calcolo della superficie e del volume del cilindro era stato risolto mediante la classe *Cilindro* che implementava l'interfaccia *SolidoRotazione*. Partiamo quindi dal seguente progetto della GUI grafica dell'applicazione.



La GUI precedente può essere realizzata dal seguente costruttore inserito nel file sorgente *ElaborazioneCiclindroGUI.java*, alla base dell'applicazione in fase di sviluppo.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class ElaborazioneCiclindroGUI extends JFrame {
	private JPanel pannello = new JPanel();
	private JLabel etichettaRaggio = new JLabel("Raggio in cm ");
	private JTextField campoRaggio = new JTextField(10);
	private JLabel etichettaAltezza = new JLabel("Altezza in cm ");
	private JTextField campoAltezza = new JTextField(10);
	private JButton pulsanteCalcola = new JButton("Calcola superficie e volume");
	private JLabel etichettaSuperficie = new JLabel();
	private JLabel etichettaVolume = new JLabel();

	public ElaborazioneCiclindroGUI() {
		this.setTitle("ElabCilindroGUI");
		this.setBounds(0, 0, 280, 200);
		this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		this.setVisible(true);
		this.pannello.add(etichettaRaggio);
		this.pannello.add(campoRaggio);
		this.pannello.add(etichettaAltezza);
		this.pannello.add(campoAltezza);
		this.pannello.add(pulsanteCalcola);
		this.pannello.add(etichettaSuperficie);
		this.pannello.add(etichettaVolume);
		this.setContentPane(this.pannello);
		this.pulanteCalcola.addMouseListener(new AscoltatoreMouse());     // Delega gestione eventi.
		this.etichettaSuperficie.setVisible(false);
```

```
                this.etichettaVolume.setVisible(false);
        }
        public static void main(String args[]) {
                ElaborazioneCiclindroGUI finestra = new ElaborazioneCiclindroGUI();
        }
}
```

Nel codice precedente, i componenti del pannello sono stati dichiarati come attributi della classe per essere visibili nell'intera applicazione. Inoltre, l'istruzione:

```
this.pulanteCalcola.addMouseListener(new AscoltatoreMouse());
```

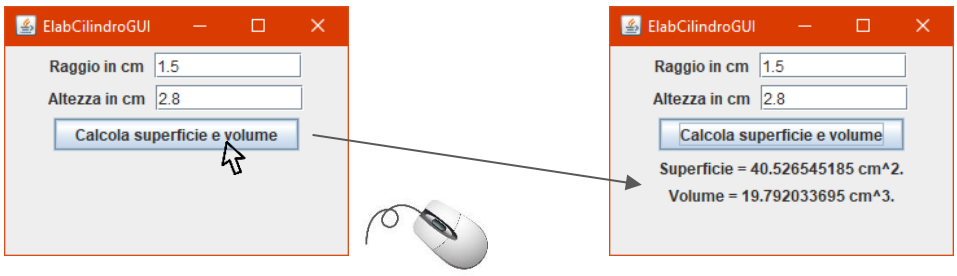
delega l'elaborazione degli eventi del mouse generati dal pulsante *pulsanteCalcola* (oggettoSorgente) a un oggetto della classe ascoltatore interna *AscoltatoreMouse*. Per terminare l'applicazione, dobbiamo completare i seguenti passi.

1. Inserire nella medesima cartella, insieme alla classe *ElaborazioneCilindroGUI.java* (e il file in bytecode *ElaborazioneCilindroGUI.class*) anche i file *Cilindro.class* e *SolidoRotazione.class*.
2. Scrivere il codice sorgente del metodo *mouseClicked()* della classe *AscoltatoreMouse*, che legge i dati digitati dall'utente nei campi di testo *raggio* e *altezza*, crea un nuovo tipo di oggetto della classe *Cilindro*, inizializzando i suoi attributi (con i valori dei campi *raggio* e *altezza*) e, infine, presenta i risultati dell'elaborazione (messaggi *oggetto.superficie()* e *oggetto.volume()*) nelle etichette *superficie* e *volume*.

Il codice sorgente dell'applicazione può essere quindi completato nel seguente modo.

```
...   // Comandi import.

public class ElaborazioneCiclindroGUI extends JFrame {
        ...   // Dichiarazione degli attributi privati.

        public ElaborazioneCiclindroGUI() {
                ...   // Corpo del costruttore.
        }   // Fine costruttore.

        public static void main(String args[]) {
                ElaborazioneCiclindroGUI finestra = new ElaborazioneCiclindroGUI();
        }

        class AscoltatoreMouse extends MouseAdapter {
                public void mouseClicked(MouseEvent oggettoEventoMouse) {
                        double valoreRaggio = Double.parseDouble(campoRaggio.getText());
                        double valoreAltezza = Double.parseDouble(campoAltezza.getText());

                        Cilindro oggettoCilindro = new Cilindro();
                        oggettoCilindro.raggio(valoreRaggio);
                        oggettoCilindro.altezza(valoreAltezza);
                        etichettaSuperficie.setText("Superficie = " + oggettoCilindro.superficie() + "
cm^2.");
                        etichettaVolume.setText("Volume = " + oggettoCilindro.volume() + " cm^3.");
                        etichettaSuperficie.setVisible(true);
                        etichettaVolume.setVisible(true);
                }   // Fine del metodo evento "mouseClicked()".
        }   // Fine della classe ascoltatore interna "AscoltatoreMouse".
}   // Fine della classe applicazione con GUI "ElaborazioneCilindroGUI".
```

Per il testing dell'applicazione è stata eseguita la seguente prova di esecuzione.



Pulsanti di opzione e caselle di selezione

I pulsanti di opzione (*radio button*) e le caselle di selezione (*check box*) sono tipi particolari di componenti che permettono all'utente della GUI di effettuare scelte in un insieme di una oppure più voci predefinite. Di fronte a un gruppo di opzioni, inserite in un pannello, l'utente può selezionare un solo pulsante di opzione alla volta, mentre può sceglierne un numero variabile tra le caselle di selezione presenti. In Java, un pulsante di opzione è un oggetto della classe JRadioButton, mentre una casella di selezione è del tipo JCheckBox. Per creare oggetti di questi due tipi di componenti, si usano costruttori del tipo seguente:

- ✓ Per i pulsanti di opzione: JRadioButton(*testo* [, *selezione*]$_{optional}$)
- ✓ per le caselle di selezione: JCheckBox(*testo* [, *selezione*]$_{optional}$)

Nei costruttori, *selezione* è un argomento booleano che vale **`true`** se si vuole far apparire il componente già in modo selezionato. L'attributo incapsulato principale dei componenti precedenti è rappresentato dal loro stato logico (**`true`** o **`false`**), che può essere gestito con i seguenti metodi.

Come fare per ...	Si impiega il metodo ...
... selezionare/deselezionare un pulsante di opzione o una casella di selezione.	setSelected(**`true`**) / setSelected(**`false`**)
... verificare se un componente è selezionato.	isSelected() restituisce il valore **`true`**

In java, per assegnare ai componenti precedenti il loro comportamento usuale, occorre raccoglierli all'interno di un gruppo che è, a sua volta, un oggetto della classe ButtonGroup. Per aggiungere un pulsante di selezione o una casella di selezione a un gruppo si impiega il metodo *add()* della classe *ButtonGroup* con la seguente sintassi generale:

oggettoButtonGroup.add(*oggettoComponente*)

L'evento principale generato da questi componenti è quello clic del mouse.

Elenchi a discesa e caselle combinate

Un elenco a discesa "a comparsa" (drop-down list) è il componente impiegato nelle GUI per l'ingresso di un dato scelto all'interno di una lista di opzioni nascosta, che appare facendo clic sul pulsante di un menu a tendina. Una casella combinata (combo box) è formato da un elenco a discesa unito a un campo di testo. La sua funzione è analoga agli elenchi a discesa, con la differenza che la voce, scelta nella lista o digitata direttamente, compare nel campo di testo abbinato all'elenco dei dati. Il Java dispone della classe JComboBox per realizzare entrambi i componenti, i cui oggetti si creano con un'istruzione della forma seguente:

JComboBox *nomeOggetto* = **new** JComboBox();

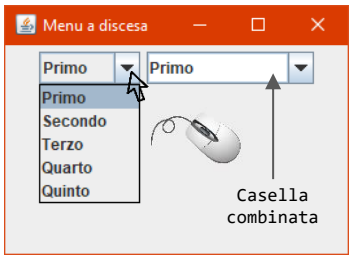
Per default, un oggetto appartenente al tipo *JComboBox* è un elenco a discesa che può essere trasformato in una determinata casella combinata richiamando il messaggio *nomeOggetto*.setEditable(**true**). L'attributo incapsulato di questi componenti è un array di oggetti generici *Object*, che può essere gestito:

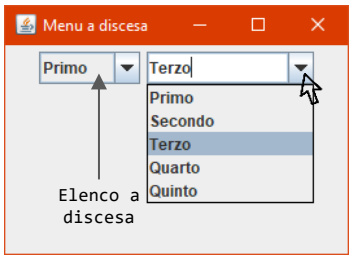
- ✓ aggiungendo un nuovo elemento con il metodo *addItem(oggetto)*;
- ✓ recuperando quelli presenti con il metodo *getItemAt(indice)*.

I principali eventi generati da questi componenti sono quelli *clic* e *doppio clic*, che avviano un'elaborazione sull'elemento scelto dall'utente in una lista di dati.

Il seguente frammento di codice sorgente, inserito nel costruttore di una finestra *JFrame*, o nel metodo *init()* di un'applet, crea un elenco a discesa e una casella combinata con dei dati esemplificativi, mostrati nella figura successiva.

```
import java.awt.*;
import javax.swing.*;

public class MenuDiscesa extends JFrame {
	public MenuDiscesa () {
		this.setTitle("Menu a discesa");
		this.setBounds(0, 0, 280, 200);
		this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		JPanel pannello = new JPanel();
		JComboBox elenco = new JComboBox();
		JComboBox combo = new JComboBox();
		combo.setEditable(true);
		elenco.addItem("Primo");
		elenco.addItem("Secondo");
		elenco.addItem("Terzo");
		elenco.addItem("Quarto");
		elenco.addItem("Quinto");
		combo.addItem("Primo");
		combo.addItem("Secondo");
		combo.addItem("Terzo");
		combo.addItem("Quarto");
		combo.addItem("Quinto");
		pannello.add(elenco);
		pannello.add(combo);
		this.setContentPane(pannello);
		this.setVisible(true);
}
	public static void main(String args[]) {
		MenuDiscesa applicazione = new MenuDiscesa();
	}
}
```





I menu

Un menu è formato da un insieme di opzioni o voci (*item*) organizzate in modo logico tra loro. A ogni opzione è associato un comando, oppure un'istruzione, che avvia una particolare elaborazione in un'applicazione o in un'applet. I menu possono essere associati a un contenitore (a discesa contestuali) oppure flottanti (pop-up). Nel seguito, per semplicità, ci riferiremo ai menu associati a un contenitore (una finestra *JFrame* o *JApplet*). Frequentemente, le voci sono organizzate in modo gerarchico in una barra dei menu formata da un insieme di menu, in cui una singola opzione potrebbe essere il punto di accesso a un altro sottomenu. In Java, una barra dei menu è un oggetto della classe JMenuBar, un menu è un'istanza del tipo JMenu, mentre una singola opzione è un oggetto della classe JMenuItem.
Per creare una barra dei menu, si segue un percorso dall'alto (la barra dei menu) verso il basso (le singole voci di menu), basato sui seguenti passi operativi.

1. Creazione di un nuovo oggetto *JMenuBar* e impostazione come barra dei menu del contenitore corrente con il metodo *setMenuBar()*, con dei comandi del tipo seguente:
   JMenuBar *oggettoBarraMenu* = **new** JMenuBar();
   this.setJMenuBar(*oggettoBarraMenu*);
2. Definizione di un'istanza dei singoli menu *JMenu* (definendo nel costruttore la voce che apparirà nella GUI) e aggiunta nella barra con il metodo *add()*;
   JMenu *oggettoMenu* = **new** JMenu(*testoVoceMenu*);
   oggettoBarraMenu.add(*oggettoMenu*);
   A un menu è possibile inserire una barra separatrice, applicando il metodo *addSeparator()* a un oggetto *JMenu*.
3. Creazione di un oggetto per ogni voce *JMenuItem* (definendo nel costruttore il testo dell'opzione nella GUI) e inserimento, con il metodo *add()*, nel menu a cui appartiene:
   JMenuItem *oggettoVoce* = **new** JMenuItem(*testoVoce*);
   oggettoMenu.add(*oggettoVoce*);

A esempio, la seguente applicazione con GUI *GestioneBarraMenu.java* crea una barra dei menu con due menu a tre voci.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class GestioneBarraDeiMenu extends JFrame {
	private JPanel pannello;
	private JMenuBar barraDeiMenu;
	private JMenu menu_1, menu_2, menu_3;
	private JMenuItem voce_A, voce_B, voce_C, voce_D, voce_E, voce_F, voce_G, voce_H, voce_I;

	public GestioneBarraDeiMenu() {
		this.setTitle("Applicazione GUI con menu");
		this.setBounds(0, 0, 280, 200);
		this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		this.pannello = new JPanel();
		this.barraDeiMenu = new JMenuBar();
		this.setJMenuBar(barraDeiMenu);
		this.menu_1 = new JMenu("Menu_1");
		this.menu_2 = new JMenu("Menu_2");
		this.menu_3 = new JMenu("Menu_3");
		this.barraDeiMenu.add(menu_1);
		this.barraDeiMenu.add(menu_2);
		this.barraDeiMenu.add(menu_3);
		this.voce_A = new JMenuItem("Voce_A");
		this.voce_B = new JMenuItem("Voce_B");
		this.voce_C = new JMenuItem("Voce_C");
		this.voce_D = new JMenuItem("Voce_D");
		this.voce_E = new JMenuItem("Voce_E");
		this.voce_F = new JMenuItem("Voce_F");
		this.voce_G = new JMenuItem("Voce_G");
		this.voce_H = new JMenuItem("Voce_H");
		this.voce_I = new JMenuItem("Voce_I");
		this.menu_1.add(voce_A); this.menu_1.add(voce_B);
		this.menu_1.add(voce_C); this.menu_2.add(voce_D);
		this.menu_2.addSeparator();
		this.menu_2.add(voce_E); this.menu_2.add(voce_F);
		this.menu_3.add(voce_G); this.menu_3.add(voce_H);
		this.menu_3.addSeparator();
		this.menu_3.add(voce_I);
```

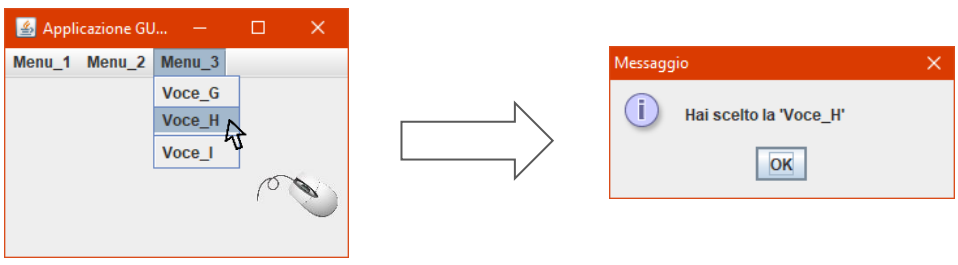
```
                this.setContentPane(pannello);
                this.setVisible(true);
        }
        public static void main(String args[]) {
                GestioneBarraDeiMenu applicazione = new GestioneBarraDeiMenu();
        }
}
```

La tabella seguente presenta alcuni metodi utili che possono essere applicati agli oggetti che rappresentano le singole voci di un menu.

Come fare per ...	Si impiega il metodo ...
... impostare/leggere il testo di una voce di un menu.	setText(*testo*) / getText()
... abilitare/disabilitare la voce di un menu.	setEnabled(**true**) / setEnabled(**false**)

L'evento principale per un menu è la scelta di una voce, che può essere gestita mediante il metodo *actionPerformed()*, dichiarato nell'interfaccia ascoltatore *ActionListener*. Questo metodo crea un evento della classe *ActionEvent*, il cui metodo *getActionCommand()* restituisce una stringa con la voce di menu scelta dall'utente. Nell'applicazione precedente, per visualizzare (in una finestra di dialogo) la scelta della voce di menu fatta dall'utente, possiamo inserire una classe ascoltatore interna delegata dai 9 oggetti *JMenuItem* (oggetti sorgente) alla gestione dell'evento *actionPerformed()*.

```
... // Comandi import.
public class GestioneBarraDeiMenu extends JFrame {
        ... // Dichiarazione attributi privati.
        public GestioneBarraDeiMenu() {
                ... // Creazione della barra dei menu.
                AscoltatoreMenu oggettoAscoltatoreMenu = new AscoltatoreMenu();
                this.voce_A.addActionListener(oggettoAscoltatoreMenu);
                this.voce_B.addActionListener(oggettoAscoltatoreMenu);
                this.voce_C.addActionListener(oggettoAscoltatoreMenu);
                this.voce_D.addActionListener(oggettoAscoltatoreMenu);
                this.voce_E.addActionListener(oggettoAscoltatoreMenu);
                this.voce_F.addActionListener(oggettoAscoltatoreMenu);
                this.voce_G.addActionListener(oggettoAscoltatoreMenu);
                this.voce_H.addActionListener(oggettoAscoltatoreMenu);
                this.voce_I.addActionListener(oggettoAscoltatoreMenu);
        }
        public static void main(String args[]) {
                GestioneBarraDeiMenu applicazione = new GestioneBarraDeiMenu();
        }   // Fine del metodo "main()".
        class AscoltatoreMenu implements ActionListener {
                public void actionPerformed(ActionEvent oggettoEventoAzione) {
                        String voceScelta = new String("");
                        voceScelta = "Hai scelto la \'" + getActionCommand() + "\'";
                        JOptionPane.showMessageDialog(null, voceScelta);
                }   // Fine del metodo evento "ActionPerformed()".
        }   // Fine della classe ascoltatore "AscoltatoreMenu".
}   // Fine della classe "GestioneBarraMenu".
```



La grafica del Java