Programmare in Java

Volume III

Esercizi

G. Frosini, G. Lettieri, A. Vecchio

6 ottobre 2015
v0.99

Indice

Prefazione

Questo volume contiene una serie di esercizi relativi al linguaggio di programmazione Java, nella versione 1.5 o superiore. Gli esercizi sono suddivisi in capitoli, dove ogni capitolo tratta di un particolare aspetto del linguaggio Java. Gli aspetti del linguaggio procedono di pari passo con la trattazione che ne viene data nei due testi "Programmare in Java, Volume I" e "Programmare in Java, Volume II" (G. Frosini e A. Vecchio). In particolare, i capitoli dal 2 al 14 corrispondono agli omonimi capitoli del testo "Programmare in Java, Volume I", mentre i capitoli dal 15 al 19 corrispondono ai capitoli del testo "Programmare in Java, Volume II". Il capitolo 1 contiene una introduzione all'utilizzo dello strumento di sviluppo "NetBeans".

Lo svolgimento degli esercizi di ogni capitolo richiede la conoscenza di tutti gli aspetti del linguaggio Java introdotti precedentemente. Nei pochi casi in cui è richiesta una conoscenza di argomenti ancora non trattati, questi vengono brevemente introdotti nel testo dell'esercizio interessato.

Per ogni esercizio viene presentata una soluzione completa. La soluzione è stata provata sull'elaboratore e successivamente inclusa nel testo, per minimizzare le possibilità di errore (ciò non esclude che qualche errore possa esservi comunque). Molte soluzioni sono corredate di note esplicative, volte a chiarire i passaggi più difficili o a sottolinerare quelli più interessanti.

Alcuni esercizi richiedono di estendere o modificare altri esercizi incontrati precedentemente nel libro.

1. Strumenti di sviluppo

1.1 Introduzione a NetBeans

NetBeans è un ambiente di sviluppo integrato per applicazioni Java. Il termine "integrato" indica la capacità dello strumento di supportare tutte le attività che ricorrono nella sviluppo di un programma: la scrittura/modifica dei file sorgenti, la loro compilazione, l'esecuzione del programma e la rimozione dei suoi malfunzionamenti (quest'ultima fase è comunemente nota come *debugging*).

NetBeans è gratuito e può essere scaricato all'indirizzo:

`http://www.netbeans.org`

Le schermate e le indicazioni riportate in questo capitolo sono basate sulla versione 5.5, l'ultima disponibile al momento della stesura.

1.2 Gestione dei progetti

Un progetto è un insieme di file sorgenti a cui sono associate alcune informazioni che l'ambiente di sviluppo usa per compilare ed eseguire il tutto. Tali informazioni includono il classpath associato alla compilazione ed all'esecuzione, gli eventuali argomenti iniziali e così via.

NetBeans include un certo numero di categorie di progetti (*General*, *Web*, *Enterprise*, *Service Oriented Architecture*, etc.) ognuna delle quali è relativa ad una particolare tipologia di applicazioni. In questo testo, per motivi di sintesi, prenderemo in considerazione la sola categoria *General*, relativa alle applicazioni Java basate sulla piattaforma Standard Edition (in altre parole, le applicazioni per macchine desktop).

All'interno della categoria *General* sono presenti alcuni modelli di progetto predefiniti, tra i più importanti citiamo:

- *Java Application* per la creazione di una normale applicazione; l'ambiente, se non indicato diversamente, crea automaticamente lo scheletro di una classe dotata di metodo *main()*.
- *Java Class Library* per la creazione di una libreria di classi prive di un metodo *main()*.
- *Java Project with Existing Sources* quando il progetto deve essere creato a partire da un insieme di file sorgenti preesistenti.

Per creare un nuovo progetto selezionare `File > New Project`: viene visualizzata una schermata che permette di scegliere la categoria e il modello del nuovo progetto (Figura 1.1a). Quindi viene visualizzata una nuova finestra con cui si può scegliere il nome del nuovo progetto, e la cartella in cui verranno contenuti i file sorgenti e binari relativi al progetto in questione (Figura 1.1b). Una volta creato, il progetto appare in una apposita lista presente sulla sinistra dello schermo.

Ogni progetto è memorizzato in una cartella all'interno della quale, oltre a file contenenti informazioni di configurazione, sono presenti le seguenti sottocartelle:

src contiene i file sorgenti; se le classi appartengono a package, al suo interno sono presenti le corrispondenti sottocartelle.

build contiene una sottocartella *classes*, all'interno della quale sono posti tutti i file *class* prodotti dalla compilazione (eventualmente strutturati in sottocartelle se le classi appartengono a package).

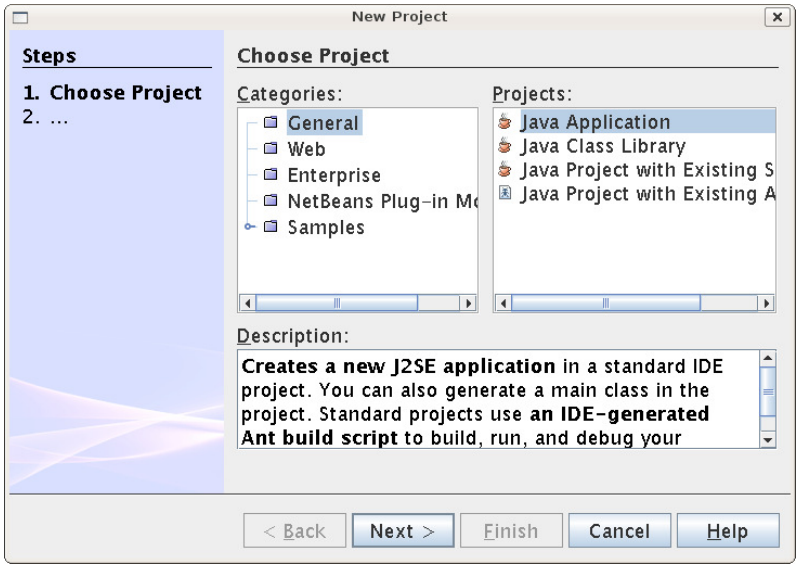
dist contiene un file archivio *jar* che può essere usato per distribuire l'applicazione agli utenti finali sotto forma di un unico file.

test contiene i file sorgenti delle classi di test (i file *class* corrispondenti sono posti in una sottocartella di *build*)

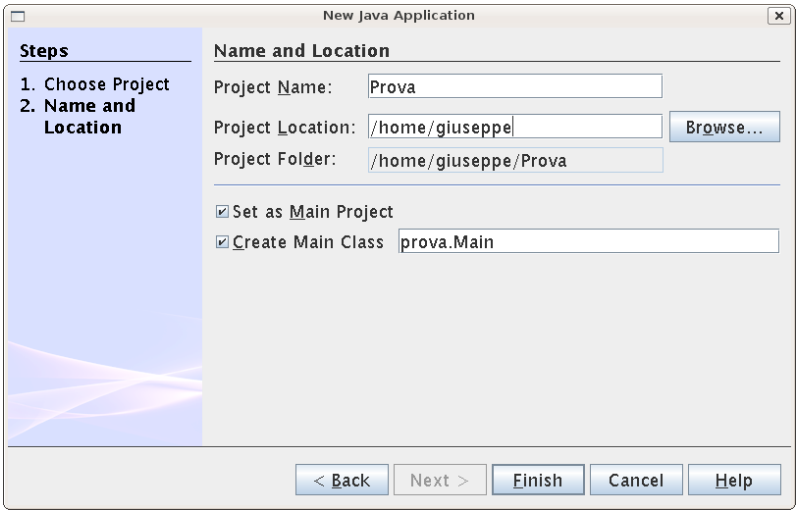
Per aprire un progetto già esistente si deve selezionare `File > Open Project`, quindi deve essere localizzata la cartella del progetto da aprire.

Per chiudere un progetto è sufficiente eseguire un click con il tasto destro del mouse sulla sua icona (sulla sinistra dello schermo), quindi si deve selezionare la voce `Close Project`.

NetBeans consente al programmatore di tenere aperti contemporaneamente tutti i progetti che vuole. Se le dimensioni di una applicazione sono particolarmente rilevanti, è possibile suddividerla in tanti progetti tra loro correlati, dove uno di essi, e solo uno, è indicato come progetto principale. La scomposizione di una applicazione in più progetti può anche essere motivata dalla volontà di favorire il riutilizzo di componenti software. Si pensi per esempio al caso in cui nello sviluppo



(a)



(b)

Figura 1.1: Creazione di un nuovo progetto

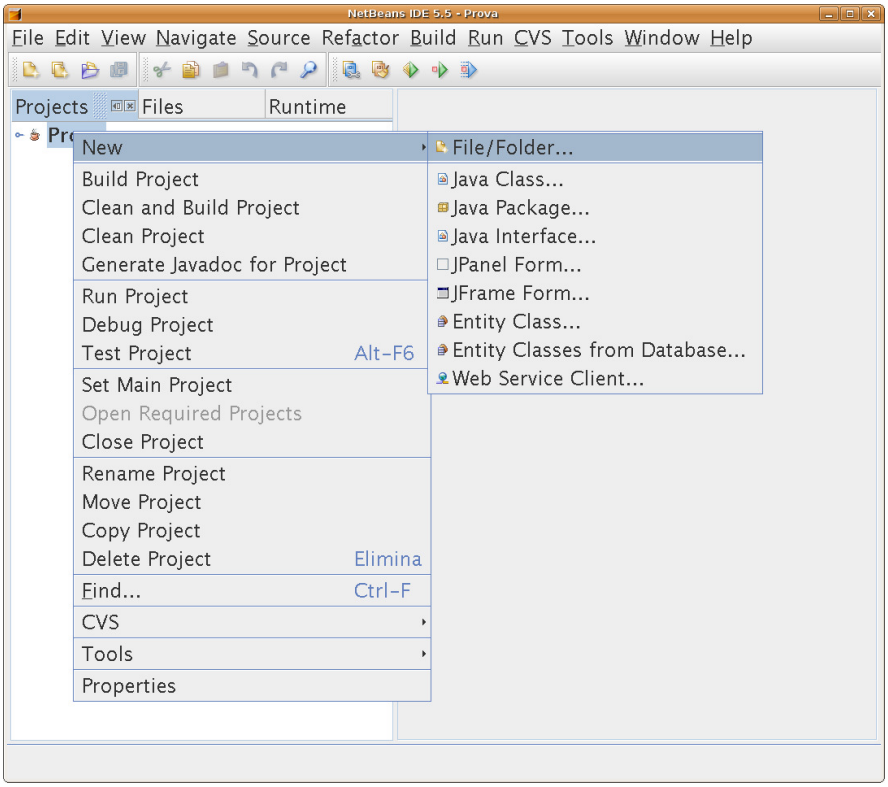


Figura 1.2: Operazioni che è possibile eseguire su un progetto

di una applicazione si renda necessaria la scrittura di una libreria: separando le due entità la libreria può essere riusata in modo semplice all'interno di altre applicazioni.

Uno dei progetti aperti può essere impostato come il progetto principale (indicato all'interno dell'ambiente di sviluppo come *Main project*): in tal caso deve contenere una classe avente un metodo *main()* che funge da punto di ingresso per l'esecuzione dell'applicazione. Per indicare un progetto come *Main project* esiste una specifica opzione da impostare alla fine della schermata che viene visualizzata quando si crea un nuovo progetto (in tal caso è anche possibile specificare il nome della classe che contiene il metodo *main()*). In alternativa, selezionare con il tasto destro del mouse il progetto dalla lista e scegliere l'opzione `Set Main project`.

Per NetBeans un progetto non contiene solo le classi e le interfacce del programma da sviluppare, ma anche classi ausiliarie da usare nella fase di test, e librerie utili allo sviluppo sia del programma che delle classi di test. Eseguendo un doppio-click

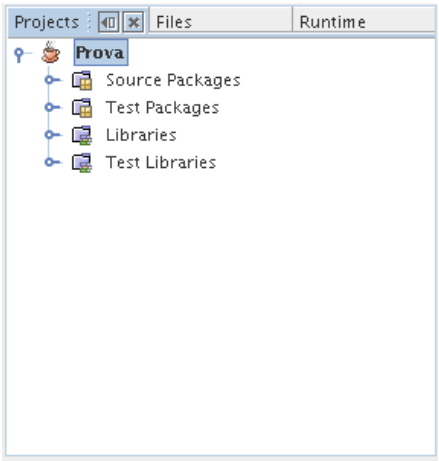


Figura 1.3: Oltre alle classi e alle interfacce del programma da sviluppare, un progetto può contenere anche classi utili per la fase di test e librerie

sull'icona di un progetto vengono visualizzate i quattro raggruppamenti di classi, come mostrato in Figura 1.3. Per semplicità supporremo che il programma da sviluppare non richieda librerie esterne e non si renda necessario l'utilizzo di classi di test o di librerie per il test.

Cliccando con il tasto destro del mouse su un progetto, viene visualizzato un menu che consente di aggiungere nuove classi. Per far questo si deve accedere al sotto-menu `New`, quindi selezionare la voce `Java Class`. Altre voci consentono di creare una nuova interfaccia o un nuovo package (Figura 1.2).

Quando si crea una nuova classe viene presentata una schermata in cui si devono specificare il nome della classe e il package di appartenenza. In Figura 1.4 viene creata una nuova classe con nome *Uno* e appartenente al package *miopkg*.

Il sistema aggiunge la nuova classe al package specificato (eventualmente creandolo), e crea lo scheletro della classe, come mostrato in Figura 1.5.

1.3 Scrittura e modifica dei file sorgenti

La maggior parte dell'area di lavoro è dedicata alla modifica dei file sorgenti. Per editare un nuovo file eseguire un doppio click sull'elemento corrispondente della lista dei progetti. L'editor include gli usuali strumenti per eseguire ricerche all'interno di un file (`Ctrl+F`) e per copiare (`Ctrl+C`), tagliare (`Ctrl+X`) e incollare (`Ctrl+P`) porzioni di testo. `Ctrl+Shift+F` riformatta il codice.

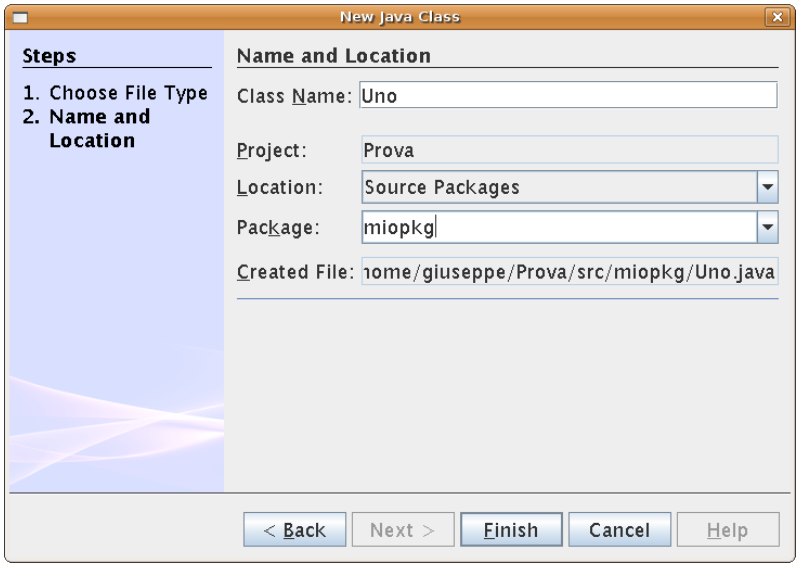


Figura 1.4: Aggiungere una classe

L'editor di NetBeans include inoltre alcune funzionalità avanzate quali il completamento automatico del codice e la visualizzazione degli errori sintattici. Il completamento automatico del codice consente al programmatore di inserire i primi caratteri del nome di un package, di una classe o di un membro di una classe, quindi di scegliere da una lista visualizzata automaticamente il nome completo. Il sistema visualizza inoltre, se disponibile, la documentazione relativa al package, alla classe o al membro (come mostrato in Figura 1.6). L'intervento del sistema di completamento del codice può essere richiesto in maniera esplicita dal programmatore usando la combinazione `Ctrl+Space`. Il sistema inoltre evidenzia alcuni errori sintattici prima ancora che il programma venga compilato. Un quadrato rosso contente una X bianca indica una riga del file sorgente che contiene un errore, e portando il puntatore del mouse su tale quadrato viene visualizzato il tipo di errore rilevato.

1.4 Compilazione

Per compilare un file, un intero package o tutto un progetto, è sufficiente eseguire un click con il tasto destro del mouse sull'elemento corrispondente della lista visualizzata sulla sinistra dello schermo, quindi scegliere la voce `Compile File`,

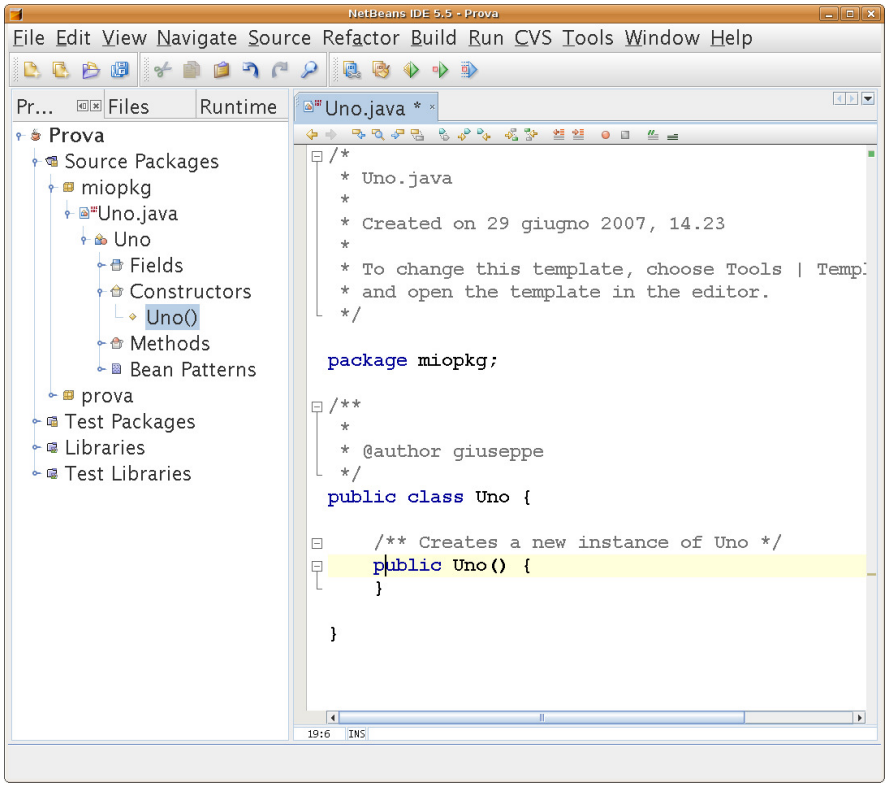


Figura 1.5: NetBeans crea automaticamente lo scheletro delle nuove classi

`Compile Package` o `Build Project`. Le medesime azioni possono essere eseguite anche attraverso il menu `Build` della barra dei menu.

I file *.class* prodotti a seguito della compilazione vengono posti nella cartella *build/classes*, contenuta all'interno della cartella del progetto. Quando si compila un intero progetto (`Build Project`), se la compilazione di tutti i file sorgenti avviene con successo, il sistema genera automaticamente anche un file archivio, avente lo stesso nome del progetto ed estensione *jar*, che contiene tutti i file *class* e può essere usato per distribuire l'applicazione[1]. Il file *jar* viene posto nella cartella *dist* contenuta all'interno di quella del progetto.

Quando si avvia la compilazione, viene mostrata una finestra nella parte bassa dello schermo che indica se l'operazione è andata a buon fine o ci sono stati degli errori. In quest'ultimo caso, con un click su un messaggio di errore si ottiene

[1] Ulteriori informazioni riguardanti i file jar possono essere trovate in Appendice A.

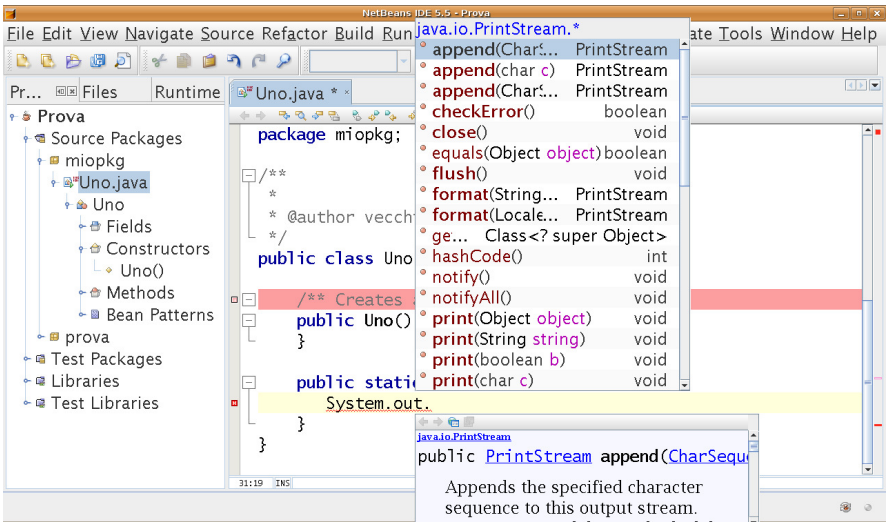


Figura 1.6: Completamento automatico del codice

l'apertura del file che contiene l'errore e la riga specifica viene messa in evidenza (Figura 1.7).

1.5 Esecuzione

Per mandare in esecuzione il progetto principale di un'applicazione scegliere `Run` dalla barra dei menu, quindi `Run Main Project`. In alternativa è possibile mandare in esecuzione un qualsiasi progetto (a patto che contenga almeno una classe dotata di un metodo *main()*) o una singola classe selezionando il corrispondente elemento dalla lista con il tasto destro e scegliendo la voce `Run Project` o `Run File`.

Per passare alla Java Virtual Machine gli argomenti iniziali cliccare con il testo destro del mouse sull'icona del progetto che deve essere eseguito, quindi scegliere la voce `Properties`. A questo punto viene mostrata una nuova finestra in cui si deve selezionare l'opzione `Run`.

I programmi in esecuzione sono visualizzati nella scheda `Runtime`. In particolare, espandendo il nodo `Processes` vengono mostrati tutti i processi in esecuzione ed è possibile arrestarli utilizzando il tasto destro del mouse (Figura 1.8).

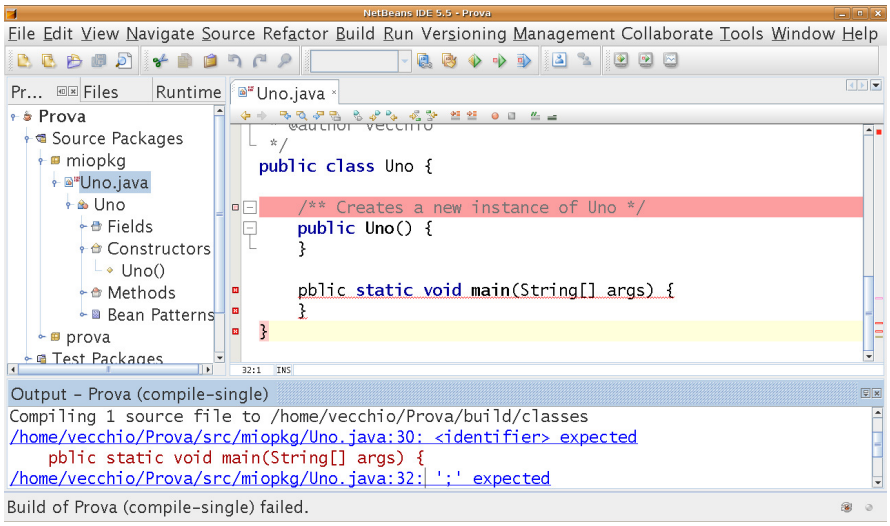


Figura 1.7: Compilazione di un file sorgente che contiene errori sintattici

1.6 Debugging

Durante una sessione di debugging il programmatore può eseguire un programma una istruzione alla volta, bloccare l'esecuzione del programma in punti specifici (detti *breakpoint*), ed esaminare lo stato del programma durante l'esecuzione. Per iniziare una sessione di debugging relativa ad un progetto (che contiene una classe dotata di metodo *main()*) o ad una classe (dotata di metodo *main()*), si deve selezionare l'elemento corrispondente dalla lista con il tasto destro del mouse e quindi scegliere `Debug Project` o `Debug Class`. L'esecuzione si arresta quando viene incontrato il primo breakpoint.

In menu `Run` contiene la maggior parte dei comandi da usare per gestire l'esecuzione controllata di un programma. In particolare: `Pause` sospende l'esecuzione, `Continue` riprende l'esecuzione, `Step Over` esegue una sola istruzione (se si tratta di una chiamata di metodo, la esegue interamente), `Step Into` esegue una sola istruzione (se si tratta di una chiamata di metodo, si blocca prima di eseguire la prima istruzione del corpo del metodo), `Step Out` esegue una sola istruzione (se l'istruzione è parte del corpo di un metodo, esegue il resto del corpo restituendo il controllo al chiamante), `Run to Cursor` esegue tutte le istruzioni comprese tra quella corrente e quella su cui è fermo il cursore.

Il modo più semplice di impostare un breakpoint è quello di eseguire un click sul bordo sinistro della riga su cui ci si vuole fermare. La presenza di un breakpoint

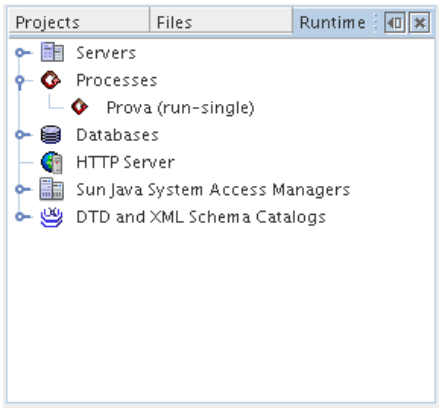


Figura 1.8: Processi in esecuzione

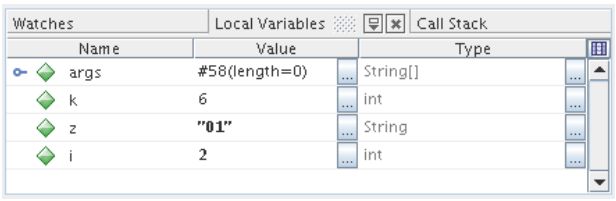


Figura 1.9: Il valore delle variabili locali durante una sessione di debugging

è indicata da una specifica icona. Per rimuovere un breakpoint è sufficiente eseguire un click sulla icona corrispondente. In alternativa, breakpoint più complessi possono essere impostati tramite il menu `Run > New Breakpoint`: viene visualizzata una schermata in cui si può scegliere il tipo di breakpoint (legato all'accesso ad una variabile, all'esecuzione di un metodo, al verificarsi di un eccezione) ed eventualmente specificare una condizione (in tal caso l'esecuzione del programma viene arrestata solo nel caso in cui la condizione risulti vera).

Quando si avvia un programma in modalità di debugging, viene automaticamente visualizzata una nuova porzione di finestra che mostra lo stato delle variabili locali visibili nel punto in cui il programma si è fermato (`Local Variables`), la lista delle chiamate fatte e non ancora terminate (`Call Stack`), e i valori delle variabili e delle espressioni che il programmatore ha indicato di voler tenere sotto osservazione (`Watches`). Per aggiungere una nuova variabile o una nuova espressione alla lista di quelle sotto osservazione, selezionare `Run > New Watch`, quindi inserire il nome della variabile o l'espressione nella apposita finestra.

1.7 La classe Console

Nei volumi precedenti è stata introdotta la classe *Console*, sviluppata dagli autori allo scopo di semplificare l'ingresso e l'uscita testuale dei dati (il file sorgente della classe *Console* può essere scaricato da `www.ing.unipi.it/LibroJava` e il suo uso è illustrato nel Capitolo 4 di "Programmare in Java, Volume I"). Per lo stesso motivo è stato introdotto il package *IngressoUscita*, che contiene una versione alternativa della classe *Console* da usare quando il programma da realizzare fa uso dei package (il package contiene inoltre alcune classi dedicate ad una gestione semplificata dell'ingresso/uscita da/verso i file). Tali classi possono essere usate anche quando lo sviluppo dei programmi avviene con NetBeans. Il programmatore può percorrere due strade. La prima, più semplice, è quella di copiare il file *Console.java* o la cartella *IngressoUscita* (con tutti i suoi file) nella cartella `src` del progetto che sta realizzando. Questa soluzione ha il vantaggio di essere estremamente semplice, ma ha il difetto di richiedere una copia del file *Console.java* (o della cartella *IngressoUscita*) per ogni progetto sviluppato. La seconda strada, più complessa, consiste nel realizzare una libreria che contiene la classe *Console* e il package *IngressoUscita*, quindi di usare tale libreria nella realizzazione dei nuovi progetti. Questa soluzione ha il vantaggio di non richiedere una copia del file *Console.java* (o della cartella *IngressoUscita*) per ogni progetto realizzato.

1.7.1 Realizzare una libreria per l'ingresso/uscita

Vediamo come realizzare una libreria per l'ingresso/uscita basata sulla classe *Console*. Il programmatore deve creare un nuovo progetto, che chiameremo *Utils*, appartenente alla categoria *General* e con modello *Java Class Library*. A questo punto deve copiare nella cartella *Utils/src* il file *Console.java* e la cartella *IngressoUscita*. Eseguendo `Build Project` l'ambiente di sviluppo produce un file archivio *Utils.jar*, posto nella cartella *Utils/dist*, che contiene la classe *Console* e le classi del package *IngressoUscita*.

Per usare la libreria così prodotta all'interno di un altro progetto, il programmatore deve cliccare con il tasto destro del mouse sull'elemento *Libraries* del progetto in questione, quindi deve scegliere la voce `Add JAR/Folder` e selezionare il file *Utils.jar* (contenuto in *Utils/dist*). Come mostrato in Figura 1.10, *Utils* diviene parte delle librerie su cui si basa il progetto (*Prova*).

Supponiamo di compilare, attraverso la voce `Build Project`, un intero progetto che usa una libreria, come per esempio la libreria *Utils*. Se l'operazione va a buon fine, nella cartella *dist* del progetto viene creato il file *jar* che contiene le classi del progetto. In più, viene creata una sottocartella *lib* che contiene tutti gli archivi delle librerie usate dal progetto. Per esempio, nel caso del progetto *Prova* che usa la libreria *Utils*, il contenuto della cartella *Prova/dist* è il seguente:

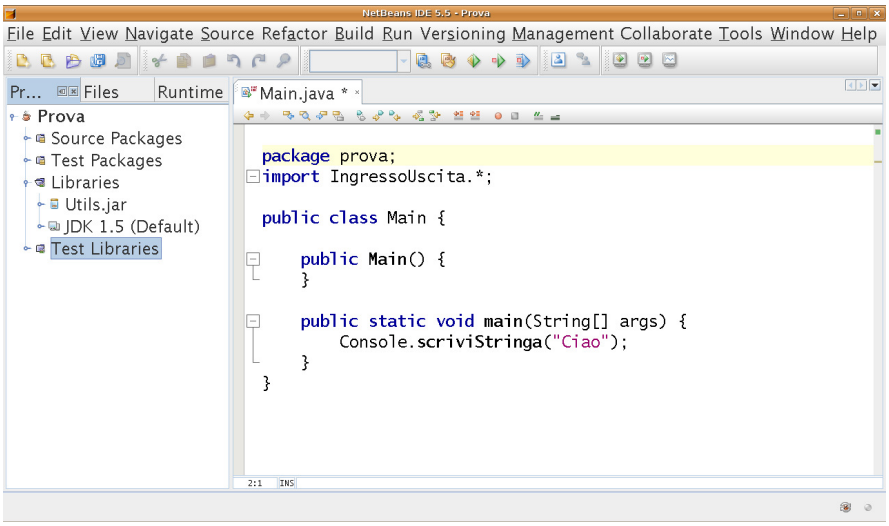


Figura 1.10: La libreria *Utils* viene usata nello sviluppo del progetto *Prova*

```
Prova/dist/
|-- Prova.jar
`-- lib
    `-- Utils.jar
```

Per distribuire l'applicazione è necessario fornire agli utilizzatori l'intero contenuto della cartella *dist*, mentre per lanciare l'applicazione è sufficiente dare il comando *java -jar Prova.jar* posizionandosi nella cartella che contiene il file *Prova.jar*.

2. Elementi di base

2.1 Espressioni

Un programma contiene le seguenti istruzioni:

```
int a = 1, b = 2, c = 10;
int r1 = a << b ^ 8;
double r2 = a / b;
double r3 = (double) a / b;
double r4 = (double) (a / b);
int r5 = c * b << a ^ c / 2;
boolean r6 = a < b & (c | b) > 6 == false;
int r7 = a++ + --b * c / b-- + 2;
```

Indicare il valore assunto dalle variabili *r1*, *r2*, ..., *r7*.

Soluzione

L'operatore di assegnamento ha la priorità più bassa e interviene per ultimo nel calcolo delle espressioni.

- L'espressione $a << b\hat{}8$ viene valutata come $(a << b)\hat{}8$. Il termine $a << b$ vale 4, in quanto è il risultato della traslazione a sinistra di due posizioni di una configurazione di bit in cui il bit meno significativo vale 1 e tutti gli altri bit sono pari a 0. A questo punto viene eseguito l'or esclusivo bit a bit tra il valore $0...0100_{due}$ e il valore $0...01000_{due}$ che produce come risultato $0...01100_{due}$. La variabile *r1* assume pertanto il valore 12.

- Poiché a e b sono interi, il termine a/b ha come valore il quoziente della divisione intera. Il quoziente della divisione, pari a 0, viene quindi convertito in double ed assegnato alla variabile *r3*, che assume quindi il valore 0.0.

- A causa della priorità degli operatori il valore di a viene prima convertito in double, quindi viene calcolato il quoziente della divisione. Essendo il dividendo un numero reale, anche il valore b viene convertito in un reale e la divisione avviene con l'aritmetica dei reali. *r3* assume quindi il valore 0.5.

- Il quoziente della divisione tra a e b viene calcolato con l'aritmetica degli interi, producendo come valore 0. Il valore 0 viene successivamente convertito in un valore reale ed assegnato alla variabile *r4*, che assume quindi il valore 0.0.

- L'espressione viene calcolata come $((c*b) << a)\hat{}(c/2)$. In particolare, il termine $((c*b) << a)$ vale 40 ($0...0101000_{due}$), mentre il termine $(c/2)$ vale 5 ($0...0101_{due}$). Il risultato dell'or esclusivo è pari a $0...0101101_{due}$, *r5* vale pertanto 45.

- L'espressione viene calcolata come $((a < b)\&(((c|b) > 6) == false))$. Il termine $(c|b)$ vale 10, pertanto $((c|b) > 6)$ vale *true*, e $(((c|b) > 6) == false))$ vale *false*. $a < b$ vale *true*, e l'and logico tra *true* e *false* produce il risultato finale *false*.

- L'espressione viene calcolata come $((a++) + (((--b)*c)/(b--)) + 2)$. Sostituendo i valori di a, b, e c: $1 + 1 * 10/1 + 2$ (gli operatori di incremento/decremento postfissi alterano il valore di una variabile dopo che nell'espressione in cui sono inseriti è stato usato il vecchio valore, inoltre la valutazione di una espressione procede da sinistra verso destra) *r7* vale 13.

2.2 Max

Siano a, b, e c tre interi. Scrivere delle espressioni per:

- calcolare il massimo tra a e b;

- calcolare il massimo tra a, b e c;

- calcolare il massimo tra a, b e c e incrementare l'oggetto con il valore più grande.

Soluzione

```
int max1 = a > b ? a : b;
int max2 =
      ((a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c));
int max3 =
      ((a > b) ? ((a > c) ? a++ : c++) : ((b > c) ? b++ : c++));
```

Note

Le parentesi hanno il solo scopo di aumentare la leggibilità del programma e non sono strettamente necessarie: l'operatore condizionale ha una priorità più bassa degli operatori di confronto e più alta dell'operatore di assegnamento.

espr1 ? espr2 : espr3 valuta solo una tra le espressioni *espr2* e *espr3*, a seconda del valore di *espr1*. Per questo motivo solo uno dei tre interi viene incrementato.

2.3 Manipola bit

Siano x e v due interi. Scrivere delle espressioni per eseguire le seguenti operazioni:

- calcolare il valore del bit v-esimo di x;
- impostare a 1 il valore del bit v-esimo di x;
- impostare a 0 il valore del bit v-esimo di x.

Soluzione

```
int b = 1 & (x >> v);
x = x | (1 << v);
x = x & ~(1 << v);
```

Note

Per risolvere il primo punto, il bit v-esimo di x viene portato nella posizione meno significativa, quindi isolato tramite l'and bit a bit con la costante 1, corrispondente ad una maschera con tutti i bit a 0, tranne il meno significativo. Nel secondo e terzo punto bisogna aver cura di lasciare inalterati tutti i bit tranne quello che si vuole modificare. Per questo motivo, nel secondo punto si usa la maschera $1 << v$, che

ha tutti i bit a 0 tranne il v-esimo, e nel terzo punto si usa la maschera $\tilde{}\,(1 << v)$, che ha tutti i bit a 1 tranne il v-esimo.

2.4 Equazioni

Scrivere una espressione Java corrispondente ad ognuna delle seguenti equazioni:

$$V = \frac{4}{3}\pi r^3 \tag{2.1}$$

$$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} \tag{2.2}$$

$$\log a^b = b\log a \tag{2.3}$$

$$e^{a+b} = e^a e^b \tag{2.4}$$

$$\sin\frac{\alpha}{2} = \sqrt{\frac{(p-b)(p-c)}{bc}} \tag{2.5}$$

$$x = \sqrt[3]{\frac{q}{2}+\sqrt{\left(\frac{p}{3}\right)^3+\left(\frac{q}{2}\right)^2}}-\sqrt[3]{-\frac{q}{2}+\sqrt{\left(\frac{p}{3}\right)^3+\left(\frac{q}{2}\right)^2}} \tag{2.6}$$

Soluzione

(2.1)
```
V = 4/3 * Math.PI * r*r*r;
```

(2.2)
```
x1 = (-b + Math.sqrt(b*b-4*a*c)) / (2*a);
```

(2.3)
```
Math.log(Math.pow(a, b)) == b * Math.log(a)
```

(2.4)
```
Math.exp(a+b) == Math.exp(a) * Math.exp(b)
```

(2.5)
```
Math.sin(alpha/2) == Math.sqrt((p-b)*(p-c)/(b*c))
```

(2.6)
```
tmp = Math.sqrt((p*p*p)/27 + q*q/4);
x = Math.pow(q/2 + tmp, 1/3) - Math.pow(-q/2 + tmp, 1/3);
```

Note

Quando si eleva ad un numero naturale piccolo (come in (2.1), (2.2) etc.), conviene, per motivi di efficienza, utilizzare il prodotto invece di ricorrere alla funzione *Math.pow()*. Si noti l'utilizzo dell'identità $\sqrt[3]{x} = x^{\frac{1}{3}}$ in (2.6).

2.5 Conversioni

Supponiamo di aver dichiarato le seguenti variabili:

```
boolean b;
int i = 20;
final int c1 = 10;
final int c2;
byte by = 200;
double d = 2.6;
```

Dire di ciascuno dei seguenti assegnamenti, che supponiamo seguano immediatamente le precedenti definizioni nel programma, se è corretto oppure no (cioè, se causa un errore a tempo di compilazione). Se un assegnamento non è corretto spiegare perché, altrimenti dire quale sarà il valore assegnato alla variabile.

```
b = i;
by = i;
by = 300;
c1 = i;
c2 = i;
i = d;
i = (int)d;
i = Math.round(d);
i = Math.round((float)d);
```

Soluzione

- L'assegnamento *b = i* è scorretto perché i tipi delle due variabili sono incompatibili.
- L'assegnamento *by = i* è scorretto, perché il valore della variabile intera *i* potrebbe non rientrare nell'intervallo di rappresentabilità di un byte. Notare che il fatto che la variabile *i* contenga, in questo caso, il valore 20, che è rappresentabile, non ha importanza.
- L'assegnamento *b = 300* è scorretto, perché il valore 300 non rientra nell'intervallo di rappresentabilità di un byte.
- L'assegnamento *c1 = i* è scorretto, perché la variabile *c1* è *final* ed è già stata assegnata una volta (al momento della dichiarazione).
- L'assegnamento *c2 = i* è corretto, perché, pur essendo *c2 final*, non le era ancora stato assegnato un valore. La variabile assumerà il valore 20.

- L'assegnamento *i = d* è scorretto, perché c'è una possibile perdita di precisione nel passaggio da *double* a *int*;
- L'assegnamento *i = (int)d* è corretto, per via della conversione esplicita ad *int*. La variabile *i* assumerà il valore 2, perché il valore 2.6 verrà troncato.
- L'assegnamento *i = Math.round(d)* causa un errore, in quanto la funzione *Math.round()* si aspetta un argomento *float*, mentre la variabile *d* è di tipo *double*, e c'è una possibile perdita di precisione nel passaggio da *double* a *float*.
- L'assegnamento *i = Math.round((float)d)* è corretto. La variabile *i* assumerà il valore 3, che l'intero più vicino al valore 2.6.

3. Istruzioni e programma

3.1 Più o meno

Scrivere un programma che legge da tastiera due reali e un carattere: se il carattere immesso è un '+' il programma mostra sul video la somma dei due reali, altrimenti la differenza.

Soluzione

PiuMeno.java:

```
1  public class PiuMeno {
2    public static void main(String[] args) {
3      Console.scriviStringa("Inserisci il primo termine");
4      double d1 = Console.leggiReale();
5      Console.scriviStringa("Inserisci il secondo termine");
6      double d2 = Console.leggiReale();
7      Console.scriviStringa("Somma (+) o sottrazione (-) ?");
8      char c = Console.leggiCarattere();
9      double r = (c == '+') ? (d1 + d2) : (d1 - d2);
10     Console.scriviStringa("Risultato:");
11     Console.scriviReale(r);
12   }
13 }
```

3.2 Potenze

Scrivere un programma che chiede di inserire un numero reale x e un numero intero n, quindi calcola e stampa il valore di x^n. Il programma non deve fare uso

della classe *Math*.

Suggerimenti

- Fare attenzione a tutti i casi particolari: esponente 0, 1, etc.
- Partire da una soluzione che funziona per esponenti positivi e generalizzarla successivamente per esponenti negativi.

Soluzione

Potenze.java:

```
1   public class Potenze {
2     public static void main(String[] args) {
3       Console.scriviStringa("Inserire la base (reale):");
4       double base = Console.leggiReale();
5       Console.scriviStringa("Inserire l'esponente (intero):");
6       int esponente = Console.leggiIntero();
7       double risultato = 1.0;
8       int valAss = (esponente > 0) ? esponente :
9                    (-esponente);
10      for (int i = 0; i < valAss; i++)
11        risultato *= base;
12      if (esponente < 0) {
13        risultato = 1.0 / risultato;
14      }
15      Console.scriviStringa("Il risultato e':");
16      Console.scriviReale(risultato);
17    }
18  }
```

Note

Per prima cosa, la soluzione calcola il valore di $r_1 = b^{|e|}$, dove b è la base ed e è l'esponente (righe 8–10). Quindi, se l'esponente era negativo, calcola $r = 1/r_1$ (righe 11–12). Si noti che, se $e = 0$, il ciclo for alle righe 9–10 non viene eseguito, e *risultato* conserva correttamente il valore 1.0, assegnatole alla riga 7.

3.3 Pensa un numero

Scrivere un programma che sceglie un numero casuale tra 1 e 100, quindi chiede all'utente di provare ad indovinarlo, per un massimo di 5 tentativi. L'utente può inserire il numero 0 per uscire prima di aver completato i tentativi. Prima dell'inserimento di ogni numero, il programma deve dire all'utente l'intervallo in cui si trova il numero da indovinare. Per aiutare l'utente, l'intervallo dovrà essere ristretto dopo ogni tentativo (il modo in cui restringerlo non è specificato). Se l'utente inserisce un numero al di fuori di tale intervallo, questo inserimento non deve essere contato come tentativo. Alla fine, sia che l'utente abbia indovinato, sia che sia abbia deciso di terminare prima, il programma deve mostrare su video il numero di tentativi utilizzati.

Soluzione

PensaNumero.java:

```
1  class PensaNumero {
2    static final int MAX_TENT = 5;
3    static final int MAX_NUM = 100;
4
5    public static void main(String[] args) {
6      int tent = 0;
7      int numero = (int) (Math.random() * MAX_NUM) + 1;
8      int min = 1;
9      int max = MAX_NUM;
10     for (;;) {
11       Console.scriviStr( "Indovina il numero che ho pensato, da");
12       Console.scriviInt(min);
13       Console.scriviStr("a");
14       Console.scriviIntero(max);
15       int n = Console.leggiIntero();
16       if (n == 0) {
17         Console.scriviStringa("Esco");
18         break;
19       }
20       if ((n < min) || (n > max)) {
21         Console.scriviStringa("Fuori intervallo");
22         continue;
23       }
24       tent++;
25       if (n == numero) {
```

```
26            Console.scriviStringa("Indovinato!");
27            break;
28          }
29          if (tent >= MAX_TENT) {
30            Console.scriviStr("Hai esaurito i tuoi");
31            Console.scriviInt(MAX_TENT);
32            Console.scriviStringa("tentativi");
33            Console.scriviStr("Il numero era");
34            Console.scriviIntero(numero);
35            return;
36          }
37          Console.scriviStringa("Ritenta");
38          if (numero > n) {
39            min = n + 1;
40          } else {
41            max = n - 1;
42          }
43        }
44        Console.scriviStr("Hai usato");
45        Console.scriviInt(tent);
46        Console.scriviStr("tentativi su");
47        Console.scriviIntero(MAX_TENT);
48      }
49    }
```

3.4 Parentesi

Scrivere un programma che legge da tastiera una sequenza di caratteri terminata dal carattere '.' e verifica se tali caratteri corrispondono ad una sequenza legale di parentesi (le parentesi sono tutte uguali, tutte tonde). Una sequenza di parentesi è legale se il numero complessivo di parentesi aperte è uguale al numero complessivo di parentesi chiuse, e se in ogni punto della sequenza non ci sono più parentesi chiuse che aperte (così ogni parentesi chiusa corrisponde ad una aperta).

Soluzione

Parentesi.java:

```
1   public class Parentesi {
2     public static void main(String[] args) {
3       int aperte = 0;
```

```
4       int c = 0;
5       Console.scriviStringa(
6         "Inserire una sequenza di parentesi");
7       Console.scriviStringa(
8         "(il carattere '.' termina la sequenza).");
9       do {
10        c = Console.leggiUnCarattere();
11        switch (c) {
12        case '(':
13          aperte++;
14          break;
15        case ')':
16          aperte--;
17          break;
18        }
19      } while ((aperte >= 0) && (c != '.'));
20      if (aperte == 0)
21        Console.scriviStringa("Sequenza legale.");
22      else {
23        Console.scriviStringa("Sequenza non legale:");
24        if (aperte > 0) {
25          Console.scriviInt(aperte);
26          Console.scriviStringa(
27            " parentesi aperte e non chiuse.");
28        } else {
29          Console.scriviStringa(
30            "una parentesi chiusa non corrisponde");
31          Console.scriviStringa(
32            "a nessuna parentesi aperta.");
33        }
34      }
35    }
36  }
```

3.5 Tavola Pitagorica

Scrivere un programma che chiede di inserire un numero positivo n da tastiera, quindi stampa su video la tavola pitagorica di ordine n. Il programma deve controllare che $n > 0$ e, in caso contrario, uscire stampando un messaggio di errore.

Soluzione

TavolaPitagorica.java:

```
1  public class TavolaPitagorica {
2    public static void main(String[] args) {
3      Console.scriviStringa("Inserire l'ordine della tavola pitagorica: ");
4      int n = Console.leggiIntero();
5      if (n <= 0) {
6        Console.scriviStringa("L'ordine deve essere maggiore di 0");
7        return;
8      }
9      for (int i = 1; i <= n; i++) {
10       for (int j = 1; j <= n; j++) {
11         Console.scriviInt((i * j));
12         Console.scriviCar('\t');
13       }
14       Console.nuovaLinea();
15     }
16   }
17 }
```

3.6 Automa

In figura 3.1 è mostrato il diagramma degli stati di un automa a stati finiti. Un automa a stati finiti è una macchina che riceve in ingresso una sequenza di valori e produce in uscita un'altra sequenza di valori. L'automa possiede uno stato interno e gli stati interni sono simboleggiati dai nodi del diagramma degli stati. Ad ogni istante, l'automa si trova nello stato corrispondente ad uno dei nodi. Ogni arco uscente dal nodo specifica il comportamento dell'automa all'arrivo del prossimo valore nella sequenza di ingresso. Per esempio, un arco etichettato con la dicitura "$A/1$" specifica che, se l'automa si trova nel nodo da cui l'arco parte e il prossimo valore in ingresso è A, l'automa deve produrre l'uscita 1. Inoltre, l'automa deve cambiare il proprio stato interno, passando al nodo puntato dall'arco.

Scrivere un programma che simula il comportamento dell'automa a stati finiti illustrato in figura 3.1.

L'automa riceve una sequenza di ingresso composta dalle lettere A e B e produce una uscita dopo ogni lettera ricevuta. A regime, l'uscita è 1 se, nelle ultime 3 lettere ricevute, comparivano più A che B, e 0 altrimenti.

Il programma, ciclicamente, deve chiedere che venga immesso un nuovo ingresso e, quindi, deve stampare su video la corrispondente uscita. Il programma termina quando l'ingresso è diverso sia da A che da B.

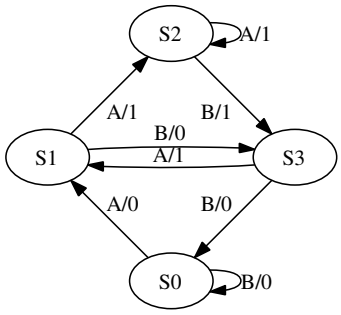


Figura 3.1: Automa a stati finiti dell'esercizio 3.6

Soluzione

Automa.java:

```
1  public class Automa {
2    public static void main(String[] args) {
3      int stato = 0;
4      int uscita = 0;
5      for (;;) {
6        Console.scriviStringa("Inserire il prossimo carattere di ingresso: ");
7        char ingresso = Console.leggiCar();
8        if ((ingresso != 'A') && (ingresso != 'B')) {
9          return;
10       }
11       switch (stato) {
12       case 0:
13         if (ingresso == 'A') {
14           uscita = 0;
15           stato = 1;
16         } else {
17           uscita = 0;
18           stato = 0;
19         }
20         break;
21       case 1:
22         if (ingresso == 'A') {
23           uscita = 1;
24           stato = 2;
25         } else {
```

```
26              uscita = 0;
27              stato = 3;
28            }
29            break;
30          case 2:
31            if (ingresso == 'A') {
32              uscita = 1;
33              stato = 2;
34            } else {
35              uscita = 1;
36              stato = 3;
37            }
38            break;
39          case 3:
40            if (ingresso == 'A') {
41              uscita = 1;
42              stato = 1;
43            } else {
44              uscita = 0;
45              stato = 0;
46            }
47            break;
48          }
49          Console.scriviStr("L'uscita e'");
50          Console.scriviIntero(uscita);
51        }
52      }
53    }
```

3.7 Sequenza di positivi

Scrivere un programma che legge in ingresso da tastiera una sequenza di numeri positivi interi e calcola il massimo, il minimo e il valore medio di tale sequenza (il programma smette di leggere altri numeri quando l'utente inserisce il valore zero o un numero negativo).

Soluzione

SequenzaNumeri.java:

```
1   public class SequenzaNumeri {
```

```
2     public static void main(String[] args) {
3       Console.scriviStringa("Inserisci una sequenza di numeri");
4       Console.scriviStringa("positivi (separati da \"invio\")");
5       Console.scriviStringa("e inserisci il valore zero");
6       Console.scriviStringa("o un numero negativo per terminare");
7       int max = 0;
8       int min = 0;
9       int somma = 0;
10      int conta = 0;
11      int x;
12      while ((x = Console.leggiIntero()) > 0) {
13        if (x > max) {
14          max = x;
15        }
16        if (conta == 0) {
17          min = x;
18        } else if (x < min) {
19          min = x;
20        }
21        somma += x;
22        conta++;
23      }
24      Console.scriviStr("Il massimo e':");
25      Console.scriviIntero(max);
26      Console.scriviStr("Il minimo e':");
27      Console.scriviIntero(min);
28      Console.scriviStr("La media e':");
29      Console.scriviReale((double) somma / conta);
30    }
31  }
```

Note

Poiché la sequenza è composta da numeri positivi, la variabile *max* viene inizializzata a zero. Successivamente è sufficiente confrontare il valore del numero letto con il valore di *max*, eventualmente aggiornando il valore di quest'ultima variabile. Analogamente, per il calcolo del minimo si potrebbe inizializzare la variabile *min* con il valore più grande rappresentabile con un *int*, quindi confrontare il valore letto con *min*, eventualmente eseguendo l'aggiornamento del valore di *min*. La soluzione proposta invece controlla se il valore letto è il primo della sequenza e in tal caso assegna il valore letto a *min*. Per tutti i valori successivi invece controlla se è neces-

sario aggiornare il valore di *min* o meno. La variabile *min* viene inizializzata con il valore 0 solo perché altrimenti verrebbe generato un errore in fase di compilazione a riga 20 (il compilatore controlla che ogni variabile abbia un valore definito prima che venga utilizzata).

4. Funzioni

4.1 Calcolo di una funzione reale

Scrivere un programma che, ciclicamente, legge da tastiera un numero reale x, calcola la funzione $y = e^x - x^2 - 2$, quindi stampa su video il valore di y.

Suggerimenti

Usare `Math.exp(x)` per ottenere e^x.

Soluzione

CalcoloFunzioneReale.java:

```
1  public class CalcoloFunzioneReale {
2    static double f(double x) {
3      return Math.exp(x) - (x * x) - 2;
4    }
5    public static void main(String[] args) {
6      double x;
7      char c;
8      do {
9        Console.scriviStringa("Inserire un numero reale: ");
10       x = Console.leggiReale();
11       Console.scriviStr("La funzione in");
12       Console.scriviReal(x);
13       Console.scriviStr("vale");
14       Console.scriviReale(f(x));
15       Console.scriviStringa("Vuoi calcolare un altro valore (s/n)?");
```

```
16          c = Console.leggiCar();
17        } while (c == 's');
18      }
19    }
```

4.2 Numeri perfetti

Scrivere una funzione *boolean perfetto(int x)* che verifica se il numero *x* è perfetto. Si ricordi che un numero è perfetto se è uguale alla somma dei suoi divisori propri e dell'unità (per esempio il numero 6, che ha come divisori propri i valori 2 e 3, e che sommati all'unità producono nuovamente il valore 6). Scrivere quindi un programma principale che stampa su video tutti i numeri perfetti inferiori ad un numero *n* letto da tastiera.

Soluzione

Perfetti.java:

```
1   public class Perfetti {
2     static boolean perfetto(int x) {
3       int somma = 1;
4       for (int i = 2; i < x; i++)
5         if ((x % i) == 0) {
6           somma += i;
7         }
8       return somma == x;
9     }
10
11    public static void main(String[] args) {
12      Console.scriviStringa("Inserisci il limite della ricerca: ");
13      int n = Console.leggiIntero();
14      for (int i = 2; i <= n; i++) {
15        if (perfetto(i)) {
16          Console.scriviStr("Il numero");
17          Console.scriviInt(i);
18          Console.scriviStringa("e' perfetto");
19        }
20      }
21    }
22  }
```

Note

Alla linea 3, si fa partire *somma* da 1, perché l'unità deve essere sempre aggiunta alla somma dei divisori propri, per definizione.

4.3 Carte di credito

Un sito di commercio elettronico deve controllare se il numero delle carte di credito usate per gli acquisti è valido o meno. Un numero valido è composto da dieci cifre, non contiene più di tre numeri pari consecutivi e non più di quattro dispari consecutivi (la cifra zero viene considerata pari). Scrivere un programma che attraverso una funzione *boolean valido(String num)* determina se il numero inserito da tastiera è valido o meno.

Soluzione

CarteDiCredito.java:

```
1  public class CarteDiCredito {
2    public static boolean valido(String num) {
3      short numPari = 0;
4      short numDisp = 0;
5      if (num.length() != 10) {
6        return false;
7      }
8      for (int i = 0; i < 10; i++) {
9        char c = num.charAt(i);
10       switch (c) {
11       case '1':
12       case '3':
13       case '5':
14       case '7':
15       case '9':
16         numPari = 0;
17         numDisp++;
18         break;
19       case '2':
20       case '4':
21       case '6':
22       case '8':
23       case '0':
```

```
24            numPari++;
25            numDisp = 0;
26            break;
27          default:
28            return false;
29          }
30          if ((numPari > 3) || (numDisp > 4)) {
31            return false;
32          }
33        }
34        return true;
35      }
36
37      public static void main(String[] args) {
38        Console.scriviStringa("Inserisci il numero di carta: ");
39        String s = Console.leggiStringa();
40        if (valido(s)) {
41          Console.scriviStringa("La carta e' buona");
42        } else {
43          Console.scriviStringa("La carta e' falsa");
44        }
45      }
46    }
```

Note

Lo scopo del ciclo 7–31 è di controllare il vincolo sul numero di cifre pari o dispari consecutive all'interno della stringa *num*. Ogni cifra pari incrementa il numero di cifre pari consecutive incontrate (*numPari*) e riazzera il numero di cifre dispari (*numDispari*) e viceversa per le cifre dispari. L'istruzione *if* alle linee 29–30 interrompe prematuramente il ciclo non appena il numero di cifre pari o dispari consecutive supera il limite previsto.

4.4 Minimo comune multiplo

Definire le funzioni *int mcm(int a, int b)* e *int mcm(int a, int b, int c)* che calcolano il minimo comune multiplo rispettivamente su due e tre interi (maggiori di 0). Scrivere quindi un programma principale che legge da tastiera due o tre interi e stampa a video il valore del loro minimo comune multiplo (chiamando l'una o l'altra funzione a seconda del numero di valori letti da tastiera).

Suggerimenti

Il minimo comune multiplo di una coppia di numeri a, b è pari a $ab/MCD(a,b)$, dove $MCD(a,b)$ è il loro massimo comune divisore.

Soluzione

MCM.java:

```
1   public class MCM {
2     public static int mcd(int a, int b) {
3       while (a != b)
4         if (a > b) {
5           a -= b;
6         } else {
7           b -= a;
8         }
9       return a;
10    }
11
12    public static int mcm(int a, int b) {
13      return (a * b) / mcd(a, b);
14    }
15
16    public static int mcm(int a, int b, int c) {
17      return mcm(mcm(a, b), c);
18    }
19
20    public static void main(String[] args) {
21      Console.scriviStringa("Quanti valori (2 o 3)?");
22      int q = Console.leggiIntero();
23      if ((q < 2) || (q > 3)) {
24        Console.scriviStringa("Valore scorretto");
25        System.exit(1);
26      }
27      Console.scriviStringa("Inserisci il primo valore");
28      int a = Console.leggiIntero();
29      Console.scriviStringa("Inserisci il secondo valore");
30      int b = Console.leggiIntero();
31      if (q == 2) {
32        Console.scriviStr("Il risultato e'");
33        Console.scriviIntero(mcm(a, b));
```

```
34          return;
35        }
36        Console.scriviStringa("Inserisci il terzo valore");
37        int c = Console.leggiIntero();
38        Console.scriviStr("Il risultato e'");
39        Console.scriviIntero(mcm(a, b, c));
40      }
41    }
```

Note

Il minimo comune multiplo di tre valori può essere calcolato determinando il minimo comune multiplo tra i primi due valori, quindi calcolando il minimo comune multiplo tra il risultato ottenuto e il terzo valore. In questo modo la funzione *int mcm(int a, int b, int c)* può essere definita semplicemente richiamando due volte la funzione *int mcm(int a, int b)*.

4.5 Conta parole

Scrivere un programma che legge una linea di caratteri da tastiera e quindi stampa il numero di *parole*, *numeri* e *parole miste* contenute nella linea. Le parole devono essere composte esclusivamente dalle lettere a–z e A–Z, i numeri esclusivamente dalle cifre 0–9, le parole miste da una combinazione di lettere e cifre. Tutti gli altri caratteri terminano il numero, la parola o la parola mista che, eventualmente, li precede, a meno che il carattere non sia preceduto da '\', nel qual caso il carattere va ignorato.

Soluzione

ContaParole.java:

```
1   public class ContaParole {
2     static final int PAROLA = 0;
3     static final int NUMERO = 1;
4     static final int MISTO = 2;
5     static final int ALTRO = 4;
6
7     static boolean isAlpha(char c) {
8       return (((c >= 'a') && (c <= 'z')) || ((c >= 'A')
9                                             && (c <= 'Z')));
10    }
```

```
11
12      static boolean isNum(char c) {
13        return ((c >= '0') && (c <= '9'));
14      }
15
16      public static void main(String[] args) {
17        String linea;
18        Console.scriviStringa("Inserire una linea");
19        linea = Console.leggiLinea();
20        int stato = ALTRO;
21        int parole = 0;
22        int numeri = 0;
23        int miste = 0;
24        linea += '.';
25        for (int i = 0; i < linea.length(); i++) {
26          char c = linea.charAt(i);
27          if (c == '\\') {
28            i++;
29            continue;
30          }
31          switch (stato) {
32          case ALTRO:
33            if (isAlpha(c)) {
34              stato = PAROLA;
35            } else if (isNum(c)) {
36              stato = NUMERO;
37            }
38            break;
39          case PAROLA:
40            if (isNum(c)) {
41              stato = MISTO;
42            } else if (!isAlpha(c)) {
43              parole++;
44              stato = ALTRO;
45            }
46            break;
47          case NUMERO:
48            if (isAlpha(c)) {
49              stato = MISTO;
50            } else if (!isNum(c)) {
51              numeri++;
```

```
52              stato = ALTRO;
53            }
54            break;
55          case MISTO:
56            if (!isAlpha(c) && !isNum(c)) {
57              miste++;
58              stato = ALTRO;
59            }
60            break;
61          }
62        }
63        Console.scriviStringa("Parole: " + parole +
64                              " Numeri: " +
65                              numeri +
66                              " Parole Miste: " + miste);
67      }
68    }
```

Note

La soluzione realizza un automa a stati finiti con 4 stati, codificati nelle costanti definite alle righe 2–5. All'inizio l'automa si trova nello stato *ALTRO*. L'automa esamina ogni carattere della riga in ingresso e, in dipendenza del tipo di carattere e del suo stato corrente, decide lo stato successivo (righe 22–58). Ogni volta che trova un carattere che non è né una lettera, né una cifra (e che, quindi termina una parola, un numero o una parola mista), incrementa il contatore opportuno (righe 38–41, 47–49 e 52–55). Le righe 24–27 provvedono a ignorare il carattere successivo ogni volta che si incontra un '\'. Notare la riga 21, che provvede ad aggiungere un carattere non alfanumerico alla fine della riga (altrimenti, una parola, numero o parola mista che terminasse alla fine della riga non verrebbe conteggiata).

4.6 Proporzione divina

Scrivere un programma che calcola attraverso una funzione ricorsiva la proporzione divina con la precisione indicata dall'utente. La proporzione divina è pari al rapporto $F(n+1)/F(n)$, dove $F(n)$ è il termine n-esimo della successione di Fibonacci, quando n tende ad infinito.

Suggerimenti

Ricordare che la successione di Fibonacci è una successione in cui ogni termine è pari alla somma dei due termini che lo precedono. I primi termini della successione sono: $1,1,2,3,5,8,13,\ldots$.

Soluzione

ProporzioneDivina.java:

```
1  public class ProporzioneDivina {
2    static double precisione;
3
4    static double phi(int s1, int s2, double precedente) {
5      double p = (double) s2 / s1;
6      if (Math.abs(p - precedente) < precisione) {
7        return p;
8      }
9      return phi(s2, s1 + s2, p);
10   }
11
12   public static void main(String[] args) {
13     Console.scriviStringa("Inserisci la precisione voluta:");
14     precisione = Console.leggiReale();
15     double pp = phi(1, 1, 0);
16     Console.scriviStringa("Proporzione divina:");
17     Console.scriviReale(pp);
18   }
19 }
```

Note

Chiamiamo $\phi_n = F(n+1)/F(n)$ l'n-esima stima della proporzione divina. L'i-esima istanza della funzione ricorsiva *phi()* (linee 3–8) riceve come argomenti $F(i)$ (variabile *s1*), $F(i+1)$ (variabile *s2*) e ϕ_{i-1} (variabile *precedente*). La funzione calcola ϕ_i (variabile *p*, linea 4) e controlla che $|\phi_i - \phi_{i-1}| < \varepsilon$ (linea 5, il valore della precisione ε è contenuto nella variabile globale *precisione*). Se la precisione desiderata non è stata raggiunta, richiama ricorsivamente se stessa con i parametri $F(i+1), F(i+2) = F(i+1) + F(i)$ e ϕ_i, altrimenti restituisce ϕ_i (che verrà restituito da tutte le istanze aperte di *phi()*, fino a raggiungere il programma principale).

5. Enumerazioni, array, stringhe

5.1 Assegnamento di array

Si consideri il seguente programma:

ArrayRef.java:

```
1   class ArrayRef {
2     public static void main(String[] args) {
3       int[] v1 = { 1, 2, 3, 4 };
4       int[] v2 = { 5, 6, 7, 8 };
5       v1 = v2;
6       v1[2] = 100;
7       Console.scriviIntero(v1[2]);
8       Console.scriviIntero(v2[2]);
9       Console.scriviIntero(v1[3]);
10    }
11  }
```

1. Dire quale sarà l'uscita del programma.

2. Cosa succede se la linea 5 viene modificata in "*System.arraycopy(v2, 0, v1, 0, 4);*"?

3. Cosa succede se la linea 3 viene modificata in "{ *int[] v1 = {1, 2};*"?

4. Cosa succede se la linea 4 viene modificata in "*int[] v2 = {5, 6, 7};*"?

Soluzione

1. Il programma stamperà i numeri 100, 100 e 8.
2. Il programma stamperà i numeri 100, 7 e 8.
3. L'uscita del programma non cambia.
4. Il programma genererà un'eccezione alla riga 9.

Note

La riga 5 originale comporta soltanto un assegnamento tra riferimenti: sia *v1* che *v2* punteranno allo stesso array. Questo spiega perchè l'assegnamento a *v1[2]* modifica anche *v2[2]*. La funzione *System.arraycopy*, invece, copia il contenuto di un array in un altro, ma i due array rimangono distinti. Nel punto 4 verrà generata un'eccezione per "accesso fuori dai limiti dell'array", in quanto *v1*, dopo l'assegnamento alla riga 5, punta ad un array che ha solo 3 elementi.

5.2 Uguaglianza tra stringhe

Si dica cosa verrà stampato a video dal seguente programma:

StringEq.java:

```
1  class StringEq {
2    public static void main(String[] args) {
3      String s1 = "Hello, world!";
4      String s2 = "Hello, world!";
5      String s3 = "Hello, ";
6      String s4 = "world!";
7      String s5;
8      Console.scriviBool(s1.equals(s2));
9      s5 = s3 + s4;
10     Console.scriviBool(s1 == s5);
11     Console.scriviBool(s1.equals(s5));
12     s5 = s1.substring(0, 7);
13     Console.scriviBool(s1 == s5);
14     Console.scriviBool(s1.equals(s5));
15     Console.scriviBool(s1 == s2);
16     Console.nuovaLinea();
17   }
18 }
```

Soluzione

Il programma stamperà:

```
true false true false false true
```

Note

Ogni variabile di tipo *String* è un riferimento ad una stringa (costante). L'operatore di uguaglianza (righe 10, 13 e 15) controlla solo l'uguaglianza tra i riferimenti e non l'uguaglianza del contenuto delle stringhe. Si noti, però, che l'uguaglianza alla riga 15 produce *true*. Ciò è dovuto al fatto che il compilatore ha memorizzato una sola volta la due stringhe "*Hello, world!*" definite alle righe 3 e 4, e vi ha fatto puntare entrambe le variabili *s1* e *s2*. Ciò è stato possibile perchè le due stringhe erano costanti (e quindi note a tempo di compilazione). Al contrario, la stringa prodotta alla riga 9, pur avendo lo stesso valore di quella puntata da *s1* e *s2*, è stata memorizzata separatamente, a tempo di esecuzione.

5.3 Zero, positivi, negativi

Definire un tipo enumerazione con nome *ZPN* caratterizzato dai valori *zero*, *positivo*, e *negativo*. Quindi, definire una funzione che prende in ingresso un array *v* di numeri interi e restituisce al chiamante un array di oggetti di tipo ZPN. L'array restituito ha la stessa dimensione di *v* e i suoi elementi hanno un valore che dipende dal valore dei corrispondenti elementi di *v* (*positivo* se l'elemento corrispondente è positivo, *zero* se l'elemento corrispondente vale 0, e così via). Infine scrivere un programma principale che chiede all'utente quanti valori vuole inserire, legge da tastiera i valori immessi dall'utente e li memorizza in un array, chiama la funzione definita in precedenza e stampa i valori ottenuti.

Soluzione

PosNeg.java:

```
1  enum ZPN {
2    ZERO, POSITIVO, NEGATIVO;
3  }
4
5  public class PosNeg {
6    public static ZPN[] determina(int[] v) {
7      ZPN[] u = new ZPN[v.length];
8      for (int i = 0; i < v.length; i++)
```

```
9         if (v[i] == 0)
10          u[i] = ZPN.ZERO;
11        else if (v[i] > 0)
12          u[i] = ZPN.POSITIVO;
13        else
14          u[i] = ZPN.NEGATIVO;
15      return u;
16    }
17    public static void stampa(ZPN[] u) {
18      for (ZPN x : u)
19        Console.scriviStringa(x.name());
20    }
21    public static void main(String[] args) {
22      Console.scriviStringa("Quanti valori?");
23      int n = Console.leggiIntero();
24      int[] vv = new int[n];
25      Console.scriviStringa("Inserisci i valori");
26      for (int i = 0; i < n; i++)
27        vv[i] = Console.leggiIntero();
28      ZPN[] uu = determina(vv);
29      stampa(uu);
30    }
31  }
```

5.4 Prodotto di polinomi

Scrivere una funzione
void prodPol (double[] p1, double[] p2, double[] r)
che, dati due polinomi di grado *n*, i cui coefficienti di grado *i* si trovano in *p1[i]* e *p2[i]* rispettivamente, calcola i coefficienti del polinomio prodotto e li inserisce nel vettore *r[]*.

Scrivere inoltre un metodo *main()* che:

1. dichiara una costante *grado* pari al grado dei polinomi dei quali si deve fare il prodotto;
2. dichiara 3 vettori *v1[]*, *v2[]*, *vr[]*, di dimensione opportuna;
3. richiede da tastiera i coefficienti dei 2 polinomi fattori e li inserisce in *v1[]* e *v2[]*;
4. chiama la funzione *prodPol()*;

5. stampa il risultato del prodotto dei polinomi;

Soluzione

ProdottoPolinomi.java:

```
1   public class ProdottoPolinomi {
2     static void prodPol(double[] p1, double[] p2,
3                         double[] r) {
4       int n = p1.length - 1;
5       for (int i = 0; i <= (2 * n); i++)
6         r[i] = 0;
7       for (int i = 0; i <= n; i++)
8         for (int j = 0; j <= n; j++)
9           r[i + j] += (p1[i] * p2[j]);
10    }
11
12    static void riempi(double[] v) {
13      int N = v.length;
14      for (int i = 0; i < N; i++)
15        v[i] = Console.leggiReale();
16    }
17
18    public static void main(String[] args) {
19      final int grado = 3;
20      double[] v1 = new double[grado + 1];
21      double[] v2 = new double[grado + 1];
22      double[] vr = new double[(2 * grado) + 1];
23      Console.scriviStringa("Inserisci i coefficienti del primo polinomio : ");
24      riempi(v1);
25      Console.scriviStringa("Inserisci i coefficienti del secondo polinomio : ");
26      riempi(v2);
27      prodPol(v1, v2, vr);
28      Console.scriviStringa("Coefficienti del polinomio prodotto: ");
29      for (int i = 0; i <= (2 * grado); i++)
30        Console.scriviStringa("Grado " + i + " : " + vr[i]);
31    }
32  }
```

5.5 Rotazione

Scrivere una funzione *void trasla(int[] v)* che ruota circolarmente verso destra di una posizione il vettore di interi passato come parametro. Quindi, scrivere un programma dotato di un metodo *main()* che:

1. crea un vettore di interi la cui dimensione è specificata dall'utente;
2. riempie il vettore con valori letti da tastiera;
3. chiede all'utente di quante posizioni vuole ruotare il vettore;
4. ruota il vettore del numero di posizioni specificato;
5. stampa il vettore risultante.

Soluzione

Rotazione.java:

```
1   public class Rotazione {
2     static void ruota(int[] vett) {
3       int n = vett.length;
4       int a = vett[n - 1];
5       for (int i = n - 1; i > 0; i--)
6         vett[i] = vett[i - 1];
7       vett[0] = a;
8     }
9
10    static void stampa(int[] vett) {
11      for (int i = 0; i < vett.length; i++)
12        Console.scriviInt(vett[i]);
13      Console.nuovaLinea();
14    }
15
16    static void riempi(int[] vett) {
17      Console.scriviStr("Inserisci");
18      Console.scriviInt(vett.length);
19      Console.scriviStringa("interi:");
20      for (int i = 0; i < vett.length; i++)
21        vett[i] = Console.leggiIntero();
22    }
23
24    public static void main(String[] args) {
```

```
25        Console.scriviStringa("Inserisci la dimensione del vettore: ");
26        int N = Console.leggiIntero();
27        int[] v = new int[N];
28        riempi(v);
29        Console.scriviStringa("Di quanti passi vuoi ruotare il vettore ?");
30        int k = Console.leggiIntero();
31        for (int j = 0; j < k; j++)
32          ruota(v);
33        stampa(v);
34      }
35    }
```

5.6 Matrice trasposta

Scrivere le seguenti funzioni.

- *void stampa (int[][] mat)*: stampa una matrice quadrata in modo incolonnato.
- *void trasponi (int[][] mat)*: traspone la matrice *mat*.

Scrivere anche un semplice *main()* che:

1. dichiara una matrice $N \times N$;
2. richiede l'immissione da tastiera degli elementi della matrice;
3. stampa la matrice;
4. richiama la funzione, passandole come parametro la matrice;
5. stampa la matrice modificata dalla funzione.

Suggerimenti

La "trasposta" di una matrice è la matrice in cui l'elemento (i, j) è scambiato di posto con l'elemento (j, i)

Es:

$$\begin{matrix} 0 & 0 & 1 \\ 2 & 9 & 3 \\ 1 & 4 & 5 \end{matrix} \rightarrow \begin{matrix} 0 & 2 & 1 \\ 0 & 9 & 4 \\ 1 & 3 & 5 \end{matrix}$$

Soluzione

Trasposta.java:

```
1   public class Trasposta {
2     static void trasponi(int[][] mat) {
3       int N = mat.length;
4       int x;
5       for (int i = 0; i < (N - 1); i++)
6         for (int j = i + 1; j < N; j++) {
7           x = mat[i][j];
8           mat[i][j] = mat[j][i];
9           mat[j][i] = x;
10        }
11    }
12
13    static void stampa(int[][] mat) {
14      int N = mat.length;
15      for (int i = 0; i < N; i++) {
16        Console.nuovaLinea();
17        for (int j = 0; j < N; j++)
18          Console.scriviStr("\t" + mat[i][j]);
19      }
20      Console.nuovaLinea();
21    }
22
23    static void riempi(int[][] mat) {
24      int N = mat.length;
25      for (int i = 0; i < N; i++)
26        for (int j = 0; j < N; j++)
27          mat[i][j] = Console.leggiIntero();
28    }
29
30    public static void main(String[] args) {
31      int N = 3;
32      int[][] m = new int[N][N];
33      Console.scriviStringa("Inizializzazione della matrice "
34                            + N
35                            + "x" + N);
36      riempi(m);
37      Console.scriviStringa("Matrice originale : ");
38      stampa(m);
```

```
39        trasponi(m);
40        Console.scriviStringa("Matrice trasposta : ");
41        stampa(m);
42      }
43    }
```

5.7 Palindroma

Scrivere una funzione che stabilisce se una stringa passata come parametro è palindroma, cioè se è identica quando viene letta da sinistra verso destra e da destra verso sinistra (esempi di stringhe palindrome: ossesso, onorarono).

Scrivere, inoltre, una seconda funzione che, nel verificare se una data stringa è palindroma, ignora le distinzioni tra maiuscole e minuscole ed esclude segni di punteggiatura e spazi (per esempio: I topi non avevano nipoti).

Il programma principale deve controllare se la prima stringa passato da riga di comando è palindroma, secondo la prima funzione, e se la stringa risultante dalla concatenazione degli argomenti passati da riga di comando è palindroma, secondo la seconda funzione.

Suggerimenti

Si ricordi che, data la stringa *s*, *s.charAt(i)* restituisce il carattere *i*-esimo.

Soluzione

Palindroma.java:

```
1   public class Palindroma {
2     static boolean palindroma1(String s) {
3       boolean ok = true;
4       int len = s.length();
5       for (int i = 0; i < (len / 2); i++) {
6         ok = s.charAt(i) == s.charAt(len - 1 - i);
7         if (!ok)
8           break;
9       }
10      return ok;
11    }
12    static boolean isAlpha(char c) {
13      return (((c >= 'a') && (c <= 'z')) ||
14              ((c >= 'A') && (c <= 'Z')));
```

```
15    }
16    static boolean palindroma2(String s) {
17      boolean ok = true;
18      s = s.toLowerCase();
19      int i = 0;
20      int j = s.length() - 1;
21      while (i < j) {
22        while ((i < s.length()) && !isAlpha(s.charAt(i)))
23          i++;
24        while ((j >= 0) && !isAlpha(s.charAt(j)))
25          j--;
26        if (i <= j)
27          ok = (s.charAt(i++) == s.charAt(j--));
28        if (!ok)
29          break;
30      }
31      return ok;
32    }
33    public static void main(String[] args) {
34      if (args.length == 0) {
35        Console.scriviStringa(
36          "Uso: java Palindroma parola (o frase)");
37        System.exit(1);
38      }
39      for (int i = 1; i < args.length; i++)
40        args[0] += (" " + args[i]);
41      Console.scriviStringa(args[0]);
42      if (palindroma1(args[0]))
43        Console.scriviStringa("La stringa e' palindroma.");
44      else if (palindroma2(args[0]))
45        Console.scriviStringa(
46          "La stringa e' palindroma" +
47          " ignorando distinzione maiuscole/minuscole" +
48          " e/o le spaziature e/o i segni di interpunzione.");
49      else
50        Console.scriviStringa(
51          "La stringa non e' palindroma.");
52    }
53  }
```

5.8 Rubrica

Scrivere un programma che ordini una rubrica di nomi. Il programma deve, ciclicamente, chiedere l'inserimento da tastiera di un nuovo nome, nella forma "nome cognome", riformattarlo internamente nella forma "cognome, nome" e, quindi, inserirlo in un vettore di nomi ordinato lessicograficamente. Si supponga che il nome termini sempre al primo carattere di spazio. Il programma deve terminare quando l'utente inserisce la parola "fine", oppure quando il vettore è pieno. Prima di terminare, il programma deve stampare su video il contenuto del vettore.

Soluzione

Rubrica.java:

```
1  class Rubrica {
2    static final int MAX = 5;
3
4    public static void main(String[] args) {
5      String[] rubrica = new String[MAX];
6      String linea;
7      int inseriti = 0;
8      for (;;) {
9        Console.scriviStringa("Inserisci 'nome cognome' (fine per terminare)");
10       linea = Console.leggiLinea();
11       if (linea.equals("fine")) {
12         break;
13       }
14       int i = 0;
15       while ((i < linea.length())
16              && (linea.charAt(i) != ' '))
17         i++;
18       if (i >= linea.length()) {
19         Console.scriviStringa("Formato scorretto");
20         continue;
21       }
22       String nuovo = linea.substring(i + 1) + ", " +
23                      linea.substring(0, i);
24       Console.scriviStringa("Nome formattato: " + nuovo);
25       i = 0;
26       while ((i < inseriti)
27              && (rubrica[i].compareTo(nuovo) < 0))
28         i++;
```

```
29          for (int j = inseriti; j > i; j--)
30            rubrica[j] = rubrica[j - 1];
31          rubrica[i] = nuovo;
32          inseriti++;
33          if (inseriti >= MAX) {
34            Console.scriviStringa("Rubrica piena!");
35            break;
36          }
37        }
38        Console.scriviStringa("Rubrica ordinata:");
39        for (int i = 0; i < inseriti; i++)
40          Console.scriviStringa(rubrica[i]);
41      }
42    }
```

Note

Il ciclo alle linee 17–20, il cui scopo è di restituire l'indice della prima occorrenza del carattere spazio all'interno della stringa *linea*, può essere eliminato facendo uso della seguente funzione della classe *String*:

int indexOf(int ch)

che ha lo stesso scopo. L'unica differenza è che la funzione *indexOf* restituisce -1 se *ch* non viene trovato all'interno della stringa, mentre il ciclo 17–20 porta la variabile *i* al valore *linea.length()*.

6. Classi

6.1 Uso dei package

Le classi che appartengono ad un package possono usare le classi che appartengono ad altri package specificandone il nome completamente qualificato (cioè il nome della classe preceduto dal package di appartenenza, usando il carattere '.' come separatore). In alternativa, è possibile importare le classi dell'altro package e successivamente utilizzare il nome semplice della classe. Per esempio, se una classe *A* appartiene al package *pack1* e vuole creare una istanza della classe *B* appartenente al package *pack2*, è sufficiente che il file *A.java* contenga il seguente codice:

```
package pack1;

public class A {
  pack2.B bb = new pack2.B();
  ...
}
```

o in alternativa:

```
package pack1;
import pack2.B;

public class A {
  B bb = new B();
  ...
}
```

Se una classe non dichiara espressamente di appartenere ad un package, essa appartiene al package senza nome. Una classe che appartiene al package senza nome

non può però essere utilizzata in altri package. Se per esempio la classe *B* avesse fatto parte del package senza nome, non sarebbe stato possibile utilizzarla nella classe A. Infatti non avremmo potuto importare il suo package in quanto privo di nome, né avremmo potuto utilizzare direttamente *B* nel codice di *A* (il nome semplice *B* sarebbe stato interpretato dal compilatore come il nome completo *pack1.B*). Da questo deriva che l'appartenenza di una classe al package senza nome costituisce un forte limite alla riusabilità della stessa, dal momento che rende impossibile l'uso della classe negli altri package.

Il lettore è invitato, a partire dal capitolo corrente e per tutti i capitoli successivi, a fare uso dei package. In particolare, suggeriamo di utilizzare uno schema che consiste nel raggruppare tutte le classi di un esercizio all'interno di un package che ha lo stesso nome dell'esercizio, tale package, a sua volta, deve essere contenuto in un package che ha il nome del capitolo. Per esempio, nell'esercizio *Dieta* si chiede di sviluppare le classi *Pasto* e *Piatto*, che quindi apparterranno al package *cap6.dieta*.

6.2 Studente

Realizzare una classe *Studente* con i seguenti metodi pubblici.

- *Studente(String nome, String matricola, int numEsami)*: costruisce un oggetto studente di nome *nome* e matricola *matricola*. Lo studente deve sostenere *numEsami* esami.
- *boolean finito()*: restituisce *true* se lo studente ha sostenuto tutti gli esami, *false* altrimenti.
- *boolean aggiungiVoto(int voto)*: lo studente ha superato un esame, con voto *voto*. Il metodo restituisce *false* quando rileva una situazione erronea, altrimenti restituisce *true*. La situazione è erronea se il metodo viene invocato quando lo studente ha già superato tutti gli esami, oppure se *voto* non è valido. Un voto valido deve essere compreso tra 18 e 30, oppure essere uguale a 33 (per indicare 30 e lode).
- *double media()*: restituisce la media dello studente. Ogni lode vale 3 punti.
- *void stampa()*: stampa nome, matricola e media dello studente sulla console.

Soluzione

Studente.java:

```
1   package cap6.studente;
2
3   import IngressoUscita.Console;
4   class Studente {
5     private String nome;
6     private String matricola;
7     private int[] voti;
8     private int nesami;
9     public Studente(String n, String m, int e) {
10      nome = n;
11      matricola = m;
12      voti = new int[e];
13    }
14    private boolean votoOK(int voto) {
15      return (voto == 33) || ((voto >= 18) && (voto <= 30));
16    }
17    public boolean finito() {
18      return nesami == voti.length;
19    }
20    public boolean aggiungiVoto(int voto) {
21      if (finito() || !votoOK(voto)) {
22        return false;
23      }
24      voti[nesami++] = voto;
25      return true;
26    }
27    public double media() {
28      int somma = 0;
29      for (int i = 0; i < nesami; i++)
30        somma += voti[i];
31      return ((double) somma) / nesami;
32    }
33    public void stampa() {
34      Console.scriviStringa("Nome: " + nome);
35      Console.scriviStringa("Matricola: " + matricola);
36      Console.scriviStringa("Media: " + media());
37    }
38  }
```

6.3 Programmi TV

Una trasmissione TV è caratterizzata da un nome e una durata (in minuti). Una programmazione consiste di una trasmissione da mandare in onda ad un certo orario. Realizzare le classi *Trasmissione* e *Programmazione* contenenti i metodi pubblici elencati di seguito.

Classe *Trasmissione*:

- *Trasmissione(String nome, int durata)*: crea un oggetto *Trasmissione* dalle caratteristiche specificate.
- *int dammiDurata()*: restituisce la durata della trasmissione.
- *String dammiNome()*: restituisce il nome della trasmissione.

Classe *Programmazione*:

- *Programmazione(Trasmissione t, int ora, int minuti)*: crea un oggetto *Programmazione*, che memorizza l'orario *ora*:*minuti* per la trasmissione *t*.
- *boolean precede(Programmazione p)*: il metodo restituisce *true* se la programmazione a cui è applicato è ad un orario precedente di quello della programmazione *p*. Restituisce *false* altrimenti.
- *boolean sovrapposto(Programmazione p)*: il metodo restituisce *true* se la programmazione a cui è applicato si sovrappone (anche parzialmente) alla programmazione *p*. Restituisce *false* altrimenti.

Soluzione

Trasmissione.java:

```
1  package cap6.programmitv;
2
3  public class Trasmissione {
4    private String nome;
5    private int durata;
6
7    public Trasmissione(String n, int d) {
8      nome = n;
9      durata = d;
10   }
11
12   public int dammiDurata() {
```

```
13      return durata;
14    }
15
16    public String dammiNome() {
17      return nome;
18    }
19  }
```

Programmazione.java:

```
1   package cap6.programmitv;
2
3   class Programmazione {
4     private Trasmissione trasm;
5     private int ora;
6     private int minuti;
7
8     public Programmazione(Trasmissione t, int h, int m) {
9       trasm = t;
10      ora = h;
11      minuti = m;
12    }
13
14    private int convertiInMinuti() {
15      return (ora * 60) + minuti;
16    }
17
18    public boolean precede(Programmazione p) {
19      return convertiInMinuti() < p.convertiInMinuti();
20    }
21
22    public boolean sovrapposto(Programmazione p) {
23      int inizio1 = convertiInMinuti();
24      int inizio2 = p.convertiInMinuti();
25      int fine1 = inizio1 + trasm.dammiDurata();
26      int fine2 = inizio2 + p.trasm.dammiDurata();
27      return (((inizio1 >= inizio2) && (inizio1 < fine2)) ||
28              ((inizio2 >= inizio1) && (inizio2 < fine1)));
29    }
30  }
```

6.4 Palestra

Una palestra prevede diverse attività, ciascuna caratterizzata da un tipo, la presenza o meno di un istruttore e un numero massimo di iscritti. È possibile cercare un certo numero di posti liberi per un certo tipo di attività, oppure per una attività qualsiasi, specificando se si vuole anche un istruttore. Realizzare le seguenti classi, con i metodi pubblici elencati di seguito.

Classe *Attivita*:

- *Attivita(String tipo, boolean istruttore, int maxIscritti)*: crea un oggetto *Attivita* dalle caratteristiche specificate.
- *int postiDisponibili()*: restituisce il numero di posti disponibili.
- *boolean iscrivi(int quante)*: iscrive *quante* persone all'attività. Restituisce *false* se l'operazione non è possibile, *true* altrimenti.
- *boolean abbandona(int quante)*: elimina l'iscrizione di *quante* persone dall'attività. Restituisce *false* se l'operazione non è possibile, *true* altrimenti.
- *String dammiTipo()*: restituisce il tipo dell'attività.
- *boolean prevedeIstruttore()*: restituisce *true* se l'attività prevede la presenza di un istruttore, *false* altrimenti.

Classe *Palestra*:

- *Palestra(int maxAttivita)*: crea un oggetto *Palestra* con le caratteristiche specificate.
- *boolean aggiungi(Attivita a)*: aggiunge *a* alle attività previste dalla palestra. Restituisce *false* se l'operazione non è possibile, *true* altrimenti.
- *Attivita cerca(Stringa tipo, boolean istruttore, int quanti)*: restituisce una attività del *tipo* specificato, per cui ci siano ancora almeno *quanti* posti disponibili. Se il *tipo* è "any", il tipo dell'attività restituita non è importante. Se *istruttore* è *true*, l'attività deve prevedere la presenza di un istruttore, altrimenti la presenza dell'istruttore non è importante. Il metodo restituisce *null* se non esiste alcuna attività con le caratteristiche richieste.

Soluzione

Attivita.java:

```
1   package cap6.palestra;
2
3   public class Attivita {
4     private boolean istruttore;
5     private int maxIscritti;
6     private int iscritti;
7     private String tipo;
8
9     public Attivita(String t, boolean i, int m) {
10      tipo = t;
11      istruttore = i;
12      maxIscritti = m;
13    }
14
15    public int postiDisponibili() {
16      return maxIscritti - iscritti;
17    }
18
19    public boolean iscrivi(int quanti) {
20      if ((iscritti + quanti) > maxIscritti) {
21        return false;
22      }
23      iscritti += quanti;
24      return true;
25    }
26
27    public boolean abbandona(int quanti) {
28      if ((iscritti - quanti) < 0) {
29        return false;
30      }
31      iscritti -= quanti;
32      return true;
33    }
34
35    public String dammiTipo() {
36      return tipo;
37    }
38
39    public boolean prevedeIstruttore() {
40      return istruttore;
41    }
```

```
42  }
```

Palestra.java:

```
1   package cap6.palestra;
2
3   public class Palestra {
4     private Attivita[] attivita;
5     private int nAttivita;
6
7     public Palestra(int maxAttivita) {
8       attivita = new Attivita[maxAttivita];
9     }
10
11    public boolean aggiungi(Attivita a) {
12      if (nAttivita >= attivita.length) {
13        return false;
14      }
15      attivita[nAttivita++] = a;
16      return true;
17    }
18
19    public Attivita cerca(String tipo, boolean istruttore,
20                          int quanti) {
21      Attivita a = null;
22      for (int i = 0; i < nAttivita; i++) {
23        a = attivita[i];
24        if (tipo.equals("any") ||
25            (tipo.equals(a.dammiTipo())
26             && (a.postiDisponibili() >= quanti) &&
27             (!istruttore || a.prevedeIstruttore()))) {
28          break;
29        }
30      }
31      return a;
32    }
33  }
```

6.5 Numeri complessi

Scrivere una classe *Complesso* che implementa un numero complesso e fornisce dei metodi per: ricavare la parte reale e la parte immaginaria del numero; ricava-

re il modulo e la fase del numero complesso; ottenere il coniugato, il reciproco e l'opposto del numero complesso; eseguire le quattro operazioni tra due numeri complessi.

Scrivere, infine, un metodo *main()* che realizza una semplice calcolatrice per numeri complessi. Il metodo deve mostrare i valore corrente di una variabile *accumulatore* (di tipo *Complesso*) e mostrare un menù tramite il quale l'utente può scegliere di: sostituire il valore corrente dell'accumulatore con il suo coniugato, il suo reciproco o il suo opposto, o con un nuovo numero complesso (che l'utente deve poter inserire da tastiera); sommargli, sottrargli, moltiplicarlo o dividerlo per un nuovo numero complesso, che l'utente deve poter inserire da tastiera; uscire dal programma. Dopo ogni operazione, il metodo deve mostrare il nuovo valore dell'accumulatore e proseguire in questo modo fino a quando l'utente non sceglie di uscire dal programma.

Soluzione

Complesso.java:

```
1  package cap6.complessi;
2
3  import IngressoUscita.Console;
4  public class Complesso {
5    private double re;
6    private double im;
7    public Complesso() {
8      re = 0;
9      im = 0;
10   }
11   public Complesso(double r, double i) {
12     re = r;
13     im = i;
14   }
15   public double getRe() {
16     return re;
17   }
18   public double getIm() {
19     return im;
20   }
21   public double getTheta() {
22     return Math.atan2(im, re);
23   }
24   public double getRho() {
```

```
25      return Math.sqrt((re * re) + (im * im));
26    }
27    public Complesso conjugate() {
28      return new Complesso(re, -im);
29    }
30    public Complesso reciprocal() {
31      double tmp = (re * re) + (im * im);
32      return new Complesso(re / tmp, -im / tmp);
33    }
34    public Complesso opposite() {
35      return new Complesso(-re, -im);
36    }
37    public Complesso add(Complesso c) {
38      return new Complesso(re + c.re, im + c.im);
39    }
40    public Complesso sub(Complesso c) {
41      return add(c.opposite());
42    }
43    public Complesso mul(Complesso c) {
44      return new Complesso(
45               (re * c.re) - (im * c.im), (re * c.im) + (im * c.re));
46    }
47    public Complesso div(Complesso c) {
48      return mul(c.reciprocal());
49    }
50    private static Complesso leggiComplesso() {
51      double re;
52      double im;
53      Console.scriviStringa("Parte reale?");
54      re = Console.leggiReale();
55      Console.scriviStringa("Parte immaginaria?");
56      im = Console.leggiReale();
57      return new Complesso(re, im);
58    }
59    public static void main(String[] args) {
60      Complesso acc = new Complesso();
61      boolean continua = true;
62      while (continua) {
63        Console.scriviStringa(
64          "Valore corrente: " + acc.getRe() + " + i*" +
65          acc.getIm());
```

```
66          Console.scriviStringa(
67            "In coordinate polari: " + acc.getRho() +
68            "*exp(i*" + acc.getTheta() + ")");
69          Console.scriviStringa("1) coniugato");
70          Console.scriviStringa("2) reciproco");
71          Console.scriviStringa("3) opposto");
72          Console.scriviStringa("4) somma");
73          Console.scriviStringa("5) sottrazione");
74          Console.scriviStringa("6) moltiplicazione");
75          Console.scriviStringa("7) divisione");
76          Console.scriviStringa("8) nuovo numero");
77          Console.scriviStringa("9) esci");
78          int scelta = Console.leggiIntero();
79          switch (scelta) {
80          case 1:
81            acc = acc.conjugate();
82            break;
83          case 2:
84            acc = acc.reciprocal();
85            break;
86          case 3:
87            acc = acc.opposite();
88            break;
89          case 4:
90            acc = acc.add(leggiComplesso());
91            break;
92          case 5:
93            acc = acc.sub(leggiComplesso());
94            break;
95          case 6:
96            acc = acc.mul(leggiComplesso());
97            break;
98          case 7:
99            acc = acc.div(leggiComplesso());
100           break;
101         case 8:
102           acc = leggiComplesso();
103           break;
104         case 9:
105           continua = false;
106           break;
```

```
107         default:
108           Console.scriviStringa(
109             "Inserire un numero da 1 a 9");
110           break;
111         }
112       }
113     }
114   }
```

6.6 Matrici quadrate

Scrivere una classe *Matrice* che implementa matrici quadrate di numeri reali, con i campi e metodi pubblici elencati di seguito.

- *final int SIZE*: dimensione delle matrici.
- *Matrice()*: costruttore, che crea una matrice di dimensioni $SIZE \times SIZE$.
- *double get(int i, int j)*: restituisce l'elemento in posizione (i, j) della matrice.
- *void set(int i, int j, double d)*: assegna il valore d all'elemento in posizione (i, j) della matrice.
- *Matrice mul(double d)*: restituisce il prodotto della matrice per lo scalare d.
- *Matrice mul(Matrice m)*: restituisce il prodotto della matrice per la matrice m.
- *double abs()*: restituisce il valore assoluto, per righe, della matrice.

Scrivere quindi una classe *Menu* che definisce un oggetto *acc* di tipo *Matrice* e, ciclicamente, mostri all'utente il contenuto dell'oggetto *acc* e un menù tramite il quale è possibile: inserire nuovi valori in *acc*; moltiplicare *acc* per uno scalare; moltiplicare *acc* per una matrice (che deve essere inserita da tastiera); stampare il valore assoluto di *acc*; uscire dal programma.

Suggerimenti

Si ricorda che il valore assoluto per righe di una matrice $A = \{a_{ij}\}$ è $max_i \sum_j |a_{ij}|$.

Per definire delle costanti usare oggetti dichiarati *static final* e interni alla classe, come visto nel capitolo relativo alle funzioni.

Soluzione

Matrice.java:

```
1  package cap6.matrici;
2
3  public class Matrice {
4    public static final int SIZE = 2;
5    private double[][] val = new double[SIZE][SIZE];
6    public void set(int i, int j, double d) {
7      val[i][j] = d;
8    }
9    public double get(int i, int j) {
10     return val[i][j];
11   }
12   public Matrice add(Matrice m) {
13     Matrice tmp = new Matrice();
14     for (int i = 0; i < SIZE; i++)
15       for (int j = 0; j < SIZE; j++)
16         tmp.val[i][j] = val[i][j] + m.val[i][j];
17     return tmp;
18   }
19   public Matrice mul(double d) {
20     Matrice tmp = new Matrice();
21     for (int i = 0; i < SIZE; i++)
22       for (int j = 0; j < SIZE; j++)
23         tmp.val[i][j] = val[i][j] * d;
24     return tmp;
25   }
26   public Matrice mul(Matrice m) {
27     Matrice tmp = new Matrice();
28     for (int i = 0; i < SIZE; i++)
29       for (int j = 0; j < SIZE; j++) {
30         tmp.val[i][j] = 0;
31         for (int k = 0; k < SIZE; k++)
32           tmp.val[i][j] += (val[i][k] * m.val[k][j]);
33       }
34     return tmp;
35   }
36   public double abs() {
37     double max = 0;
38     for (int i = 0; i < SIZE; i++) {
```

```
39        double work = 0;
40        for (int j = 0; j < SIZE; j++)
41          work += Math.abs(val[i][j]);
42        if (work > max)
43          max = work;
44      }
45      return max;
46    }
47  }
```

Menu.java:

```
1   package cap6.matrici;
2
3   import IngressoUscita.Console;
4   public class Menu {
5     private static void stampaMatrice(Matrice m) {
6       for (int i = 0; i < m.SIZE; i++) {
7         for (int j = 0; j < m.SIZE; j++)
8           Console.scriviStr(m.get(i, j) + " ");
9         Console.nuovaLinea();
10      }
11    }
12    private static Matrice leggiMatrice() {
13      Matrice m = new Matrice();
14      for (int i = 0; i < m.SIZE; i++)
15        for (int j = 0; j < m.SIZE; j++) {
16          Console.scriviStringa("Elemento (" + i + ", " + j +
17                                ")?");
18          m.set(i, j, Console.leggiReale());
19        }
20      return m;
21    }
22    public static void main(String[] args) {
23      Matrice acc = new Matrice();
24      boolean ancora = true;
25      while (ancora) {
26        stampaMatrice(acc);
27        Console.scriviStringa("1) nuova matrice");
28        Console.scriviStringa("2) moltiplica per scalare");
29        Console.scriviStringa("3) moltiplica per matrice");
30        Console.scriviStringa("4) stampa valore assoluto");
```

```
31          Console.scriviStringa("5) esci");
32          switch (Console.leggiIntero()) {
33          case 1:
34            acc = leggiMatrice();
35            break;
36          case 2:
37            Console.scriviStringa("Inserire scalare");
38            acc = acc.mul(Console.leggiReale());
39            break;
40          case 3:
41            acc = acc.mul(leggiMatrice());
42            break;
43          case 4:
44            Console.scriviStringa("Valore assoluto: " +
45                                  acc.abs());
46            break;
47          case 5:
48            ancora = false;
49            break;
50          }
51        }
52      }
53    }
```

6.7 Matrici complesse

Modificare la classe *Matrici* dell'esercizio 6.6 in modo da realizzare matrici quadrate di numeri complessi, invece che di numeri reali. Riutilizzare la classe *Complesso* dell'esercizio 6.5.

Suggerimenti

Si ricorda che, se $x = a + ib$ è un numero complesso, $|x| = \sqrt{a^2 + b^2}$.

Per definire delle costanti usare oggetti dichiarati *static final* e interni alla classe, come visto nel capitolo relativo alle funzioni.

Soluzione

MatriceComplessa.java:

```
1   package cap6.matrcompl;
```

```
2
3   import cap6.complessi.Complesso;
4   public class MatriceComplessa {
5     public static final int SIZE = 2;
6     private Complesso[][] val = new Complesso[SIZE][SIZE];
7     public MatriceComplessa() {
8       for (int i = 0; i < SIZE; i++)
9         for (int j = 0; j < SIZE; j++)
10          val[i][j] = new Complesso();
11    }
12    public void set(int i, int j, Complesso d) {
13      val[i][j] = d;
14    }
15    public Complesso get(int i, int j) {
16      return val[i][j];
17    }
18    public MatriceComplessa add(MatriceComplessa m) {
19      MatriceComplessa tmp = new MatriceComplessa();
20      for (int i = 0; i < SIZE; i++)
21        for (int j = 0; j < SIZE; j++)
22          tmp.val[i][j] = val[i][j].add(m.val[i][j]);
23      return tmp;
24    }
25    public MatriceComplessa mul(Complesso d) {
26      MatriceComplessa tmp = new MatriceComplessa();
27      for (int i = 0; i < SIZE; i++)
28        for (int j = 0; j < SIZE; j++)
29          tmp.val[i][j] = val[i][j].mul(d);
30      return tmp;
31    }
32    public MatriceComplessa mul(MatriceComplessa m) {
33      MatriceComplessa tmp = new MatriceComplessa();
34      for (int i = 0; i < SIZE; i++)
35        for (int j = 0; j < SIZE; j++) {
36          tmp.val[i][j] = new Complesso();
37          for (int k = 0; k < SIZE; k++)
38            tmp.val[i][j] = tmp.val[i][j].add(val[i][k].mul(
39                                                  m.val[k][j]));
40        }
41      return tmp;
42    }
```

```
43    public double abs() {
44      double max = 0;
45      for (int i = 0; i < SIZE; i++) {
46        double work = 0;
47        for (int j = 0; j < SIZE; j++)
48          work += val[i][j].getRho();
49        if (work > max) {
50          max = work;
51        }
52      }
53      return max;
54    }
55  }
```

Note

Notare che la linea 7 crea soltanto una matrice di riferimenti nulli a oggetti di tipo *Complesso*. Per completare l'inizializzazione di un oggetto di tipo *MatriceComplessa*, è necessario anche inizializzare ciascuno di quei riferimenti con un nuovo oggetto *Complesso* (linee 9–11).

6.8 Dieta

Un piatto è caratterizzato da un nome, da informazioni nutrizionali, e da un tipo. I tipi di piatti possibili sono *primo*, *secondo*, *contorno* e *frutta*. Le informazioni nutrizionali di interesse sono relative alla quantità di grassi, di proteine e di carboidrati presenti (memorizzate come *float*). Un pasto si compone di più piatti, in particolare ci può essere al più un piatto per ogni tipo, mentre alcuni tipi di piatti possono essere assenti (in un pasto potrebbero essere presenti il primo, il contorno e la frutta, mentre il secondo potrebbe non essere presente). *Tipo* può essere realizzato come un tipo enumerazione. Scrivere le classi *InfoNutriz*, *Piatto*, e *Pasto* in modo che siano dotate almeno dei seguenti metodi e/o costruttori.

Classe *InfoNutriz*:

- *InfoNutriz(float c, float g, float p)*: crea un oggetto in cui le quantità di carboidrati, grassi e proteine sono pari rispettivamente a c, g, p.

Classe *Piatto*:

- *Piatto(String n, Tipo t, InfoNutriz i)*: crea un piatto avente nome n, tipo t, e informazioni nutrizionali pari a i.

Classe *Pasto*:

- *boolean aggiungi(Piatto p)*: se possibile aggiunge il piatto *p* al pasto. Restituisce il valore *true* se il piatto viene aggiunto (non era stato aggiunto un piatto dello stesso tipo in precedenza), *false* altrimenti.
- *InfoNutriz getInfo()*: restituisce una istanza di *InfoNutriz* relativa all'intero pasto.
- *boolean equilibrato()*: restituisce il valore *true* se il pasto contiene sia la frutta che il contorno ed almeno uno tra il primo ed il secondo.
- *void stampa()*: stampa a video il nome di tutti i piatti, il loro tipo, e le informazioni nutrizionali associate ad ognuno di essi.

Scrivere infine un programma principale di prova in cui viene creato un pasto contenente alcuni piatti, quindi vengono stampate le sue caratteristiche.

Suggerimenti

Dato un tipo enumerazione, ad ogni enumeratore è associato un valore intero che per default vale 0 per il primo enumeratore, 1 per il secondo e così via. Tale valore può essere ottenuto invocando il metodo ***int ordinal()*** sull'enumeratore.

Soluzione

InfoNutriz.java:

```
1  package cap6.dieta;
2
3  public class InfoNutriz {
4    private float carbo;
5    private float grassi;
6    private float prot;
7
8    public InfoNutriz() {
9    }
10
11   public InfoNutriz(float c, float g, float p) {
12     carbo = c;
13     grassi = g;
14     prot = p;
15   }
```

```
16
17    public void aggiungi(InfoNutriz s) {
18      carbo += s.carbo;
19      grassi += s.grassi;
20      prot += s.prot;
21    }
22
23    public String comeStringa() {
24      return "carbo.=" + carbo + ", grassi=" + grassi +
25             ", prot.="
26             + prot;
27    }
28  }
```

Tipo.java:

```
1   package cap6.dieta;
2
3   public enum Tipo {
4     PRIMO, SECONDO, CONTORNO, FRUTTA
5   }
```

Piatto.java:

```
1   package cap6.dieta;
2
3   public class Piatto {
4     private Tipo t;
5     private InfoNutriz in;
6     private String nome;
7
8     public Piatto(String n, Tipo tt, InfoNutriz i) {
9       nome = n;
10      t = tt;
11      in = i;
12    }
13
14    public Tipo getTipo() {
15      return t;
16    }
17
18    public InfoNutriz getInfo() {
```

```
19      return in;
20    }
21
22    public String getNome() {
23      return nome;
24    }
25
26    public String comeStringa() {
27      return t.name() + ": " + nome + " " +
28             in.comeStringa();
29    }
30  }
```

Pasto.java:

```
1   package cap6.dieta;
2
3   import IngressoUscita.Console;
4   public class Pasto {
5     private Piatto[] pp = new
6     Piatto[Tipo.values().length];
7     public boolean aggiungi(Piatto p) {
8       if (pp[p.getTipo().ordinal()] == null) {
9         pp[p.getTipo().ordinal()] = p;
10        return true;
11      }
12      return false;
13    }
14    public InfoNutriz getInfo() {
15      InfoNutriz ii = new InfoNutriz();
16      for (Piatto p : pp)
17        if (p != null)
18          ii.aggiungi(p.getInfo());
19      return ii;
20    }
21    public boolean equilibrato() {
22      return ((pp[Tipo.FRUTTA.ordinal()] != null) &&
23              (pp[Tipo.CONTORNO.ordinal()] != null) &&
24              ((pp[Tipo.PRIMO.ordinal()] != null) ||
25               (pp[Tipo.SECONDO.ordinal()] != null)));
26    }
27    void stampa() {
```

```
28      for (Piatto p : pp)
29        if (p != null)
30          Console.scriviStringa(p.comeStringa());
31    }
32  }
```

Prova.java:

```
1   package cap6.dieta;
2
3   import IngressoUscita.Console;
4   class Prova {
5     public static void main(String[] args) {
6       Piatto p1 =
7         new Piatto(
8         "Spaghetti alla carbonara", Tipo.PRIMO,
9         new InfoNutriz(50.0f, 15.6f, 32.3f));
10      Piatto p2 =
11        new Piatto(
12        "Cinghiale in umido", Tipo.SECONDO,
13        new InfoNutriz(34.5f, 24.9f, 23.5f));
14      Piatto p3 =
15        new Piatto(
16        "Mela", Tipo.FRUTTA,
17        new InfoNutriz(9.4f, 7.5f, 1.4f));
18      Pasto pas = new Pasto();
19      if (
20        pas.aggiungi(p1) && pas.aggiungi(p2) &&
21        pas.aggiungi(p3))
22        Console.scriviStringa("Ho aggiunto tutti i piatti");
23      else
24        Console.scriviStringa(
25          "Non sono riuscito ad aggiungere un piatto");
26      if (pas.equilibrato())
27        Console.scriviStringa("Bravo");
28      else
29        Console.scriviStringa("Alimentazione scorretta");
30      Console.scriviStringa(
31        "Informazioni nutrizionali (intero pasto):");
32      Console.scriviStringa(pas.getInfo().comeStringa());
33      pas.stampa();
34    }
```

```
35  }
```

6.9 Archivio immobiliare

Un archivio immobiliare gestisce i dati di un insieme di immobili. Ogni immobile è caratterizzato da indirizzo, numero civico, superficie, prezzo, e numero di vani. Realizzare le classi *Immobile* e *ArchivioImm* come segue:

Classe *Immobile*:

- *Immobile(String indir, int numCiv, float sup, float prez, int numVani)*: crea un oggetto *Immobile* dalle caratteristiche specificate.
- *float prezzoAlMq()*: restituisce il prezzo al metro quadro dell'immobile.
- *void stampa()*: stampa i dati dell'immobile.
- Eventuali metodi accessori che consentano di accedere alle caratteristiche dell'oggetto.

Classe *ArchivioImm*:

- *ArchivioImm(int n)*: crea un archivio in grado di contenere i dati di al più *n* immobili.
- *boolean inserisci(Immobile i)*: inserisce l'immobile specificato nell'archivio; restituisce true se l'immobile viene inserito correttamente, *false* altrimenti (l'archivio è pieno).
- *boolean elimina(String in, int nc)*: rimuove, se presente, l'immobile che ha indirizzo *in* e numero civico *nc*. Restituisce *true* se l'immobile era presente in archivio, *false* altrimenti.
- *Immobile conveniente()*: restituisce l'immobile con il prezzo al metro quadro più basso presente in archivio (*null* se l'archivio è vuoto).
- *float prezzoMedio(int v)*: restituisce il prezzo medio degli immobili contenuti nell'archivio che hanno *v* vani.
- *void stampa()*: stampa il contenuto dell'archivio.

Soluzione

Immobile.java:

```
1  package cap6.archivioimm;
2
3  import IngressoUscita.Console;
4  public class Immobile {
5    private String indir;
6    private int numCiv;
7    private float mq;
8    private float prezzo;
9    private int vani;
10   public Immobile(String i, int n, float m, float p,
11                   int v) {
12     indir = i;
13     numCiv = n;
14     mq = m;
15     prezzo = p;
16     vani = v;
17   }
18   public float costoAlMq() {
19     return prezzo / mq;
20   }
21   public String dammiIndir() {
22     return indir;
23   }
24   public int dammiNumCiv() {
25     return numCiv;
26   }
27   public int dammiVani() {
28     return vani;
29   }
30   public float dammiPrezzo() {
31     return prezzo;
32   }
33   public void stampa() {
34     Console.scriviStringa(indir + ", " + numCiv + ", " +
35                           mq +
36                           ", " + prezzo +
37                           ", " + vani);
38   }
```

```
39  }
```

ArchivioImm.java:

```
1   package cap6.archivioimm;
2
3   import IngressoUscita.Console;
4   public class ArchivioImm {
5     private Immobile[] imm;
6     private int numImm;
7     public ArchivioImm(int n) {
8       imm = new Immobile[n];
9     }
10    public boolean inserisci(Immobile i) {
11      if (numImm < imm.length) {
12        imm[numImm] = i;
13        numImm++;
14        return true;
15      }
16      return false;
17    }
18    public boolean elimina(String in, int nc) {
19      for (int i = 0; i < numImm; i++)
20        if (imm[i].dammiIndir().equals(in)
21            && (imm[i].dammiNumCiv() == nc)) {
22          imm[i] = imm[--numImm];
23          imm[numImm] = null;
24          return true;
25        }
26      return false;
27    }
28    public Immobile conveniente() {
29      Immobile conv = null;
30      if (numImm == 0) {
31        return conv;
32      }
33      conv = imm[0];
34      for (int i = 1; i < numImm; i++)
35        if (imm[i].costoAlMq() < conv.costoAlMq()) {
36          conv = imm[i];
37        }
38      return conv;
```

```
39      }
40      public float prezzoMedio(int v) {
41        int conta = 0;
42        float s = 0.0f;
43        for (int i = 0; i < numImm; i++)
44          if (imm[i].dammiVani() == v) {
45            s += imm[i].dammiPrezzo();
46            conta++;
47          }
48        if (conta != 0) {
49          s /= conta;
50        }
51        return s;
52      }
53      public void stampa() {
54        for (int i = 0; i < numImm; i++)
55          imm[i].stampa();
56      }
57    }
```

6.10 Biblioteca

Una biblioteca gestisce un insieme di libri. Ogni libro è caratterizzato da un titolo, un autore e una data di pubblicazione. Un autore è caratterizzato da un nome e un cognome. Realizzare le classi *Data*, *Libro*, *Autore* e *Biblioteca* in modo che sia possibile compiere le azioni elencate di seguito.

Classe *Data*:

- *Data(int giorno, int mese, int anno)*: crea un oggetto *Data* dalle caratteristiche specificate.
- *int confronta(Data d)*: confronta l'oggetto *Data* a cui viene applicato con la data *d*, e restituisce un numero negativo, zero, o un numero positivo rispettivamente quando l'oggetto precede, è uguale a, o segue la data *d*.

Classe *Autore*:

- *Autore(String n, String c)*: crea un oggetto *Autore* con nome *n* e cognome *c*.
- *boolean uguale(Autore a)*: restituisce true se il nome e il cognome dell'oggetto implicito sono uguali a quelli dell'oggetto *a*; il valore *null* per il nome e/o il cognome dell'autore *a* può essere usato come valore "jolly"; in tutti gli altri casi restituisce *false*.

Classe *Libro*:

- *Libro(String t, Autore a, Data d)*: crea un oggetto *Libro* dalle caratteristiche specificate.
- *String getTitolo()*: restituisce il titolo del libro.
- *Autore getAutore()*: restituisce l'autore del libro.
- *Data getDataPub()*: restituisce la data di pubblicazione del libro.

Classe *Biblioteca*:

- *Biblioteca(int n)*: costruttore che crea un oggetto *B*iblioteca in grado di gestire al più *n* libri.
- *boolean aggiungiLibro(Libro l)*: aggiunge il libro *l* a quelli gestiti dalla biblioteca; restituisce *true* se è possibile aggiungere il libro, *false* altrimenti (la biblioteca contiene già il numero massimo di libri).
- *Libro[] cercaPerAutore(Autore a)*: restituisce i libri che hanno l'autore specificato.
- *Libro[] cercaPerCognome(String c)*: restituisce i libri il cui autore ha il cognome specificato.
- *Libro[] elenco()*: restituisce l'elenco di tutti i libri.
- *Libro[] cercaRecenti(Data d)*: restituisce l'elenco dei libri la cui data di pubblicazione è più recente di *d*.
- *boolean elimina(String t)*: se un libro con titolo *t* è presente nella biblioteca lo elimina e restituisce il valore *true*, *false* altrimenti.

Soluzione

Libro.java:

```
1  package cap6.biblio;
2
3  public class Libro {
4    private String titolo;
5    private Autore autore;
6    private Data dataPub;
7
8    public Libro(String t, Autore a, Data d) {
```

```
9       titolo = t;
10      autore = a;
11      dataPub = d;
12    }
13
14    public Autore getAutore() {
15      return autore;
16    }
17
18    public String getTitolo() {
19      return titolo;
20    }
21
22    public Data getDataPub() {
23      return dataPub;
24    }
25  }
```

Autore.java:

```
1   package cap6.biblio;
2
3   import IngressoUscita.Console;
4   public class Autore {
5     private String nome;
6     private String cognome;
7     public Autore(String n, String c) {
8       nome = n;
9       cognome = c;
10    }
11    public boolean uguale(Autore a) {
12      if ((a.nome == null) && (a.cognome == null)) {
13        return true;
14      }
15      if (a.nome == null) {
16        return cognome.equals(a.cognome);
17      }
18      if (a.cognome == null) {
19        return nome.equals(a.nome);
20      }
21      return nome.equals(a.nome)
22             && cognome.equals(a.cognome);
```

```
23    }
24    public void stampa() {
25      Console.scriviStringa(nome + " " + cognome);
26    }
27  }
```

Data.java:

```
1   package cap6.biblio;
2
3   import IngressoUscita.Console;
4   public class Data {
5     private int giorno;
6     private int mese;
7     private int anno;
8     public Data(int g, int m, int a) {
9       giorno = g;
10      mese = m;
11      anno = a;
12    }
13    public int confronta(Data d) {
14      if (anno != d.anno) {
15        return anno - d.anno;
16      }
17      if (mese != d.mese) {
18        return mese - d.mese;
19      }
20      return giorno - d.giorno;
21    }
22    public void stampa() {
23      String s = giorno + "/" + mese + "/" + anno;
24      Console.scriviStringa(s);
25    }
26  }
```

Biblioteca.java:

```
1   package cap6.biblio;
2
3   public class Biblioteca {
4     private Libro[] lib;
5     private int numLibri;
```

```
6
7     public Biblioteca(int n) {
8       lib = new Libro[n];
9     }
10
11    public boolean aggiungiLibro(Libro l) {
12      if (numLibri < lib.length) {
13        lib[numLibri++] = l;
14        return true;
15      } else {
16        return false;
17      }
18    }
19
20    public Libro[] cercaPerAutore(Autore a) {
21      int n = 0;
22      for (int i = 0; i < numLibri; i++)
23        if (lib[i].getAutore().uguale(a)) {
24          n++;
25        }
26      if (n == 0) {
27        return null;
28      }
29      Libro[] risul = new Libro[n];
30      n = 0;
31      for (int i = 0; i < numLibri; i++)
32        if (lib[i].getAutore().uguale(a)) {
33          risul[n++] = lib[i];
34        }
35      return risul;
36    }
37
38    public Libro[] cercaPerCognome(String c) {
39      return cercaPerAutore(new Autore(null, c));
40    }
41
42    public Libro[] elenco() {
43      return cercaPerAutore(new Autore(null, null));
44    }
45
46    public Libro[] cercaRecenti(Data d) {
```

```
47        int n = 0;
48        for (int i = 0; i < numLibri; i++)
49          if (lib[i].getDataPub().confronta(d) >= 0) {
50            n++;
51          }
52        if (n == 0) {
53          return null;
54        }
55        Libro[] risul = new Libro[n];
56        n = 0;
57        for (int i = 0; i < numLibri; i++)
58          if (lib[i].getDataPub().confronta(d) >= 0) {
59            risul[n++] = lib[i];
60          }
61        return risul;
62      }
63
64      public boolean elimina(String t) {
65        for (int i = 0; i < numLibri; i++)
66          if (lib[i].getTitolo().equals(t)) {
67            lib[i] = lib[--numLibri];
68            lib[numLibri] = null;
69            return true;
70          }
71        return false;
72      }
73    }
```

Note

Notare che il metodo *uguale()* della classe *Autore* ci permette di scrivere una sola volta il codice di ricerca per autore (linee 17–30) e di riutilizzarlo anche nella ricerca per cognome e nell'elenco, usando opportuni valori jolly (linee 32 e 35).

6.11 Labirinto

Un labirinto è composto da stanze e porte che collegano le stanze tra loro. Un giocatore, ad ogni istante, si trova in una stanza del labirinto e può decidere di attraversare una porta per passare in un altra stanza. Scrivere le classi *Risultato*, *Stanza*, *Porta* e *Giocatore*, che permettono di realizzare un labirinto. La classe *Risultato* serve a comunicare l'esito di un tentativo di spostamento. La classe *Stanza* contiene

una descrizione e può essere collegata a quattro porte (una per ogni punto cardinale). La classe *Porta* è associata a due stanze e permette ad un giocatore di passare da una stanza ad un'altra. Infine, la classe *Giocatore* rappresenta il giocatore. Le classi devono contenere almeno i metodi elencati di seguito.

Classe *Risultato*:

- *Risultato(boolean ris, String mess)*: costruisce un *Risultato* che indica successo o fallimento (parametro *ris*) e contiene un messaggio che descrive il motivo del successo o del fallimento (parametro *mess*).
- *boolean isOk()*: restituisce *true* se il movimento ha avuto successo, *false* altrimenti.
- *String getMessage()*: restituisce la stringa che motiva il successo/fallimento.

Classe *Stanza*:

- *Direzioni*: una opportuna enumerazione.
- *Stanza(String des)*: costruisce una stanza descritta da *des* e non collegata ad alcuna porta.
- *void collega(Direzioni dir, Porta porta)*: collega a *porta* il muro in direzione *dir* della stanza.
- *Risultato vai(Giocatore giocatore, Direzioni direzione)*: prova a spostare *giocatore* nella *direzione* specificata. Se la direzione è collegata ad una porta e lo spostamento ha successo, la funzione restituisce un oggetto *Risultato* contenente i valori *true* e "Fatto.". Se la direzione non è collegata a nessuna porta o lo spostamento non ha successo, il giocatore non viene spostato e la funzione restituisce un oggetto *Risultato* contenente il valore *false* e un messaggio che spieghi il motivo dell'insuccesso.
- *String descrivi()*: restituisce la descrizione della stanza.
- *Direzioni[] direzioni()*: restituisce un array contenente tutte e sole le direzioni della stanza che risultano collegate ad una porta.

Classe *Porta*:

- *Porta(Stanza stanza1, Stanza stanza2)*: costruisce una porta associata a *stanza1* e *stanza2*.
- *Risultato attraversa(Giocatore giocatore)*: se *giocatore* si trova in *stanza1*, lo sposta in *stanza2* e viceversa. In entrambi i casi restitiuisce un oggetto *Risultato* contenente i valori *true* e "Fatto.". Se *giocatore* non si trova né in *stanza1*, né in *stanza2* non sposta *giocatore* e restituisce un oggetto *Risultato* contenente il valore *false* e un messaggio di errore.

Classe *Giocatore*:

- *Giocatore(String nome)*: costruisce un giocatore di nome *nome* che inizialmente non si trova in nessuna stanza.
- *void muovi(Stanza stanza)*: posiziona il giocatore nella stanza passata come argomento.
- *Stanza locazione()*: restituisce la stanza in cui il giocatore si trova attualmente.

Scrivere quindi un programma di test che costruisca un labirinto con almeno tre stanze e permetta all'utente di spostarsi tra le stanze. Il programma deve creare un oggetto *Giocatore* e, ciclicamente, mostrare la descrizione della stanza in cui il giocatore si trova; deve permettere all'utente di inserire una stringa, controllare che tale stringa sia "nord", "sud", "est", "ovest" o "fine" e comportarsi di conseguenza.

Soluzione

Risultato.java:

```
1  package cap6.labirinto;
2
3  public class Risultato {
4    private boolean stato;
5    private String messaggio;
6
7    public Risultato() {
8      stato = true;
9      messaggio = "Fatto.";
10   }
11
12   public Risultato(boolean stato, String messaggio) {
13     this.stato = stato;
14     this.messaggio = messaggio;
15   }
16
17   public boolean isOk() {
18     return stato;
19   }
20
21   public String getMessage() {
22     return messaggio;
23   }
24 }
```

Stanza.java:

```
1  package cap6.labirinto;
2
3  public class Stanza {
4    public enum Direzioni {
5      NORD, SUD, EST, OVEST;
6    }
7    protected String descr;
8    protected Porta[] porte = new Porta[4];
9    public Stanza(String descr) {
10     this.descr = descr;
11   }
12   public void collega(Direzioni dir, Porta porta) {
13     porte[dir.ordinal()] = porta;
14   }
15   public Risultato vai(Giocatore giocatore,
16                        Direzioni dir) {
17     if (porte[dir.ordinal()] == null)
18       return new Risultato(
19                 false, "Non puoi andare in quella direzione");
20     else
21       return porte[dir.ordinal()].attraversa(giocatore);
22   }
23   public String descrivi() {
24     return descr;
25   }
26   public Direzioni[] direzioni() {
27     int num = 0;
28     for (Direzioni d : Direzioni.values())
29       if (porte[d.ordinal()] != null)
30         num++;
31     Direzioni[] dir = new Direzioni[num];
32     int i = 0;
33     for (Direzioni d : Direzioni.values())
34       if (porte[d.ordinal()] != null) {
35         dir[i] = d;
36         i++;
37       }
38     return dir;
39   }
40 }
```

Porta.java:

```
1  package cap6.labirinto;
2
3  public class Porta {
4    protected Stanza stanza1;
5    protected Stanza stanza2;
6
7    public Porta(Stanza stanza1, Stanza stanza2) {
8      this.stanza1 = stanza1;
9      this.stanza2 = stanza2;
10   }
11
12   public Risultato attraversa(Giocatore giocatore) {
13     Stanza stanza = giocatore.locazione();
14     if (stanza == stanza1) {
15       giocatore.muovi(stanza2);
16     } else if (stanza == stanza2) {
17       giocatore.muovi(stanza1);
18     } else {
19       return new Risultato(false, "Errore nella mappa");
20     }
21     return new Risultato();
22   }
23 }
```

Giocatore.java:

```
1  package cap6.labirinto;
2
3  public class Giocatore {
4    protected Stanza stanza;
5    protected String nome;
6
7    public Giocatore(String nome) {
8      this.nome = nome;
9    }
10
11   public void muovi(Stanza stanza) {
12     this.stanza = stanza;
13   }
14
15   public Stanza locazione() {
```

```
16      return stanza;
17    }
18  }
```

Test.java:

```
1   package cap6.labirinto;
2
3   import IngressoUscita.Console;
4   class Test {
5     public static void main(String[] args) {
6       Stanza s1 = new
7       Stanza("Stai pilotando il tuo monoposto " +
8              "sopra la desolata regione scozzese delle Highlands");
9       Stanza s2 = new Stanza("Ti trovi in un campo aperto,"
10                             +
11                             " a ovest di una casa bianca," +
12                             " con la porta principale sbarrata");
13      Stanza s3 = new
14      Stanza("Sei in un labirinto di tortuosi," +
15             " piccoli passaggi, tutti simili");
16      Porta p1 = new Porta(s1, s2);
17      s1.collega(Stanza.Direzioni.EST, p1);
18      s2.collega(Stanza.Direzioni.OVEST, p1);
19      Porta p2 = new Porta(s2, s3);
20      s2.collega(Stanza.Direzioni.SUD, p2);
21      s3.collega(Stanza.Direzioni.NORD, p2);
22      Giocatore giocatore = new Giocatore("Zak");
23      giocatore.muovi(s1);
24      boolean ancora = true;
25      while (ancora) {
26        Stanza s = giocatore.locazione();
27        Console.scriviStringa(s.descrivi());
28        Stanza.Direzioni[] dir = s.direzioni();
29        if (dir.length > 0) {
30          Console.scriviStringa("Puoi andare a:");
31          for (int i = 0; i < dir.length; i++)
32            Console.scriviStringa(dir[i].toString());
33        }
34        Console.scriviStringa("Dove vuoi andare?");
35        String in = Console.leggiStringa();
36        if (in.equals("fine")) {
```

```
37            ancora = false;
38          } else if (in.equals("nord") || in.equals("sud")
39                     || in.equals("est") ||
40                     in.equals("ovest")) {
41            Stanza.Direzioni d = Stanza.Direzioni.valueOf(
42                                 in.toUpperCase());
43            Risultato r = s.vai(giocatore, d);
44            Console.scriviStringa(r.getMessage());
45          } else {
46            Console.scriviStringa("Non capisco la parola " + in);
47          }
48        }
49      }
50    }
```

Note

Alla linea 46 del file *Test.java* si utilizza la funzione membro statica predefinita ***Direzioni valueOf(String s)*** che converte una stringa nell'omonimo valore dell'enumerazione. La funzione è *case sensitive* e da ciò deriva la necessità di convertire preventivamente la stringa *in* in maiuscolo tramite la funzione membro *toUpperCase()* della classe *String*.

6.12 Package

Supponiamo di avere le seguenti classi:

A.java:

```
1   package pack1;
2
3   class A {
4     void m1() {}
5     void m2() {
6       B b = new B();
7       b.m1();
8     }
9   }
```

B.java:

```
1   package pack1;
```

```
2
3   class B {
4     B() {
5     }
6
7     void m1() {
8     }
9   }
```

C.java:

```
1   package pack1;
2
3   public class C {
4     void m1() {
5       A a = new A();
6       a.m1();
7     }
8   }
```

S.java:

```
1   package pack2;
2
3   import pack1.B;
4   public class S {
5     private void m1() {
6       A a = new A();
7       a.m1();
8       B b = new B();
9       b.m1();
10      C c = new C();
11    }
12  }
```

Quali errori vengono visualizzati in fase di compilazione? E come è possibile correggerli?

Soluzione

Il comando di compilazione produce le seguenti segnalazioni:

errori:

```
1   S.java:2: pack1.B is not public in pack1; cannot be accessed
2             from outside package
3   import pack1.B;
4                ^
5   S.java:7: cannot find symbol
6   symbol  : class A
7   location: class pack2.S
8     { A a = new A();
9       ^
10  S.java:7: cannot find symbol
11  symbol  : class A
12  location: class pack2.S
13    { A a = new A();
14                 ^
15  S.java:9: cannot find symbol
16  symbol  : class B
17  location: class pack2.S
18      B b = new B();
19      ^
20  S.java:9: cannot find symbol
21  symbol  : class B
22  location: class pack2.S
23      B b = new B();
24                 ^
25  5 errors
```

che derivano dai seguenti errori:

- righe 1-3: la classe *S* appartiene al package *pack2* mentre la classe *B* appartiene al package *pack1*. L'istruzione *import pack1.B* produce una segnalazione di errore in quanto la classe *B* non è pubblica e pertanto non è visibile all'esterno del package *pack1*. Non essendo stato trovato il tipo *B*, vengono prodotte gli errori indicati alle righe 14-22. Per risolvere questo problema è necessario che la classe *B* venga definita pubblica (`public class B ...`).

- righe 5-13: la classe *S* non importa la classe *A* e pertanto il tipo *A* non è definito. Per risolvere questo problema è necessario aggiungere ad *S* l'istruzione `import pack1.A` e definire pubblica la classe *A* (`public class A ...`).

Una volta apportate queste modifiche, se si prova a compilare le classi in questione si ottengono dei nuovi messaggi di errore:

errori:

```
S.java:9: m1() is not public in pack1.A; cannot be accessed
          from outside package
    a.m1();
     ^
S.java:10: B() is not public in pack1.B; cannot be accessed
           from outside package
    B b = new B();
          ^
S.java:11: m1() is not public in pack1.B; cannot be accessed
           from outside package
    b.m1();
     ^
3 errors
```

che hanno i seguenti motivi:

- Il metodo *m1()* della classe *A* non è pubblico e la classe *S* che tenta di usarlo non appartiene allo stesso package di *A*.
- Il costruttore della classe *B* non è pubblico e la classe *S* che tenta di usarlo non appartiene allo stesso package di *A*.
- Il metodo *m1()* della classe *B* non è pubblico e la classe *S* che tenta di usarlo non appartiene allo stesso package di *B*.

Per poter compilare correttamente le classi è necessario quindi modificarle come segue:

A.java:

```
package pack1;

public class A
{ public void m1()
  { //...
  }
  void m2()
  { B b = new B();
    b.m1();
  }
}
```

B.java:

```
1   package pack1;
2
3   public class B
4   { public B()
5     { //...
6     }
7     public void m1()
8     { //...
9     }
10  }
```

C.java:

```
1   package pack1;
2
3   public class C
4   { void m1()
5     { A a = new A();
6       a.m1();
7     }
8   }
```

S.java:

```
1   package pack2;
2   import pack1.A;
3   import pack1.B;
4   import pack1.C;
5
6   public class S
7   { private void m1()
8     { A a = new A();
9       a.m1();
10      B b = new B();
11      b.m1();
12      C c = new C();
13    }
14  }
```

7. Liste

7.1 Lista doppia

Realizzare una classe *ListaDoppia* come una lista doppiamente collegata di oggetti di tipo *Elemento* (dove *Elemento* deve essere una classe statica definita all'interno di *ListaDoppia*). Ogni oggetto di tipo *Elemento* contiene un campo *info* (di tipo intero), un riferimento all'oggetto che lo precede e un riferimento all'oggetto che lo segue nella lista. La classe *ListaDoppia* deve fornire i seguenti metodi:

- *void inTesta(int info)*: inserisce un nuovo *Elemento*, contenente il valore *info*, in testa alla lista.
- *void inCoda(int info)*: inserisce un nuovo *Elemento*, contenente il valore *info*, in coda alla lista.
- *boolean estrai(int info)*: estrae dalla lista il primo elemento contenente il valore *info*. Restituisce vero o falso a seconda se l'elemento richiesto è stato trovato, o meno, nella lista.

Scrivere quindi un programma di test che crei una lista doppia e, ciclicamente, ne mostri il contenuto, permettendo all'utente di scegliere se inserire (in testa o in coda) un nuovo elemento, o di estrarre un elemento scelto dall'utente stesso, oppure di uscire dal programma.

Soluzione

ListaDoppia.java:

```
1  package cap7.listadoppia;
2
```

```
3   public class ListaDoppia {
4     private Elemento testa;
5     private Elemento coda;
6
7     public void inTesta(int info) {
8       Elemento e = new Elemento(info);
9       e.next = testa;
10      testa = e;
11      if (e.next != null) {
12        e.next.prev = e;
13      } else {
14        coda = e;
15      }
16    }
17
18    public void inCoda(int info) {
19      Elemento e = new Elemento(info);
20      e.prev = coda;
21      coda = e;
22      if (e.prev != null) {
23        e.prev.next = e;
24      } else {
25        testa = e;
26      }
27    }
28
29    public boolean estrai(int info) {
30      Elemento scorri = testa;
31      while ((scorri != null) && (scorri.info != info))
32        scorri = scorri.next;
33      boolean trovato = scorri != null;
34      if (trovato) {
35        if (scorri.next != null) {
36          scorri.next.prev = scorri.prev;
37        } else {
38          coda = scorri.prev;
39        }
40        if (scorri.prev != null) {
41          scorri.prev.next = scorri.next;
42        } else {
43          testa = scorri.next;
```

```
44          }
45        }
46        return trovato;
47      }
48
49      public String toString() {
50        Elemento scorri = testa;
51        String s = "";
52        while (scorri != null) {
53          s += (scorri.info + " ");
54          scorri = scorri.next;
55        }
56        return s;
57      }
58
59      static class Elemento {
60        int info;
61        Elemento next;
62        Elemento prev;
63
64        Elemento(int i) {
65          info = i;
66        }
67      }
68    }
```

Test.java:

```
1   package cap7.listadoppia;
2
3   import IngressoUscita.Console;
4   class Test {
5     public static void main(String[] args) {
6       ListaDoppia l = new ListaDoppia();
7       for (;;) {
8         Console.scriviStringa("Lista: " + l.toString());
9         Console.scriviStringa("Operazioni:");
10        Console.scriviStringa("1) Inserimento in testa");
11        Console.scriviStringa("2) Inserimento in coda");
12        Console.scriviStringa("3) Estrazione");
13        Console.scriviStringa("4) Esci");
14        int op = Console.leggiIntero();
```

```
15          if ((op < 1) || (op > 4)) {
16            Console.scriviStringa("Scelta non valida");
17            continue;
18          }
19          if (op == 4) {
20            Console.scriviStringa("Ciao");
21            break;
22          }
23          Console.scriviStringa("Inserire un numero");
24          int num = Console.leggiIntero();
25          switch (op) {
26          case 1:
27            l.inTesta(num);
28            break;
29          case 2:
30            l.inCoda(num);
31            break;
32          case 3:
33            if (l.estrai(num)) {
34              Console.scriviStringa("OK");
35            } else {
36              Console.scriviStringa("Non trovato");
37            }
38            break;
39          default:
40            break;
41          }
42        }
43      }
44    }
```

7.2 Rubrica 2

Risolvere nuovamente l'esercizio 5.8, ma utilizzando una lista al posto di un vettore. Si rimuova, di conseguenza, la limitazione sul numero massimo di nomi da inserire. (Per semplicità, si tralasci la parte dell'esercizio che richiede la riformattazione della stringa inserita dall'utente).

Soluzione

Rubrica2.java:

```
1  class Elemento {
2    String nome;
3    Elemento next;
4  }
5
6  class Rubrica2 {
7    public static void main(String[] args) {
8      Elemento rubrica = null;
9      for (;;) {
10       Console.scriviStringa(
11         "Inserisci 'nome' (fine per terminare)");
12       String nuovo = Console.leggiLinea();
13       if (nuovo.equals("fine"))
14         break;
15       Elemento scorri = rubrica;
16       Elemento prec = null;
17       while (
18         (scorri != null) &&
19         (scorri.nome.compareTo(nuovo) < 0)) {
20         prec = scorri;
21         scorri = scorri.next;
22       }
23       Elemento tmp = new Elemento();
24       tmp.nome = nuovo;
25       tmp.next = scorri;
26       if (prec != null)
27         prec.next = tmp;
28       else
29         rubrica = tmp;
30     }
31     Console.scriviStringa("Rubrica ordinata:");
32     for (
33       Elemento scorri = rubrica; scorri != null;
34       scorri = scorri.next)
35       Console.scriviStringa(scorri.nome);
36   }
37 }
```

7.3 Ruota

Una ruota è divisa in spicchi, ciascuno dei quali può essere opaco o trasparente. Le operazioni che possono essere effettuate su una ruota sono le seguenti:

- *Ruota()*: costruttore di default, che crea una ruota *r* formata da un solo spicchio. Tale spicchio è trasparente.
- *Ruota(int n)*: costruttore che crea una ruota formata da *n* spicchi. Tali spicchi sono trasparenti.
- *void rendiOpaco(int s)*: rende opaco lo *s*-esimo spicchio della ruota.
- *void rendiTrasparente(int s)*: rende trasparente lo *s*-esimo spicchio della ruota.
- *boolean seiOpaco(int s)*: restituisce *true* se lo *s*-esimo spicchio è opaco e *false* altrimenti.
- *void gira(int p)*: gira la ruota in avanti di *p* posizioni (cioè, il primo spicchio diventa il *(p + 1)*-esimo, il secondo spicchio diventa il *(p + 2)*-esimo e così via).
- *void sovrapponi(Ruota r)*: modifica la ruota sovrapponendo a ciascuno dei suoi spicchi il corrispondente spicchio della ruota *r*. La sovrapposizione di due spicchi è trasparente se ambedue tali spicchi sono trasparenti, altrimenti la sovrapposizione è opaca. Le due ruote devono avere lo stesso numero di spicchi.
- *void spicchi(int m)*: modifica la ruota in modo tale che essa diventi composta da *m* spicchi. Se inizialmente essa ha più di *m* spicchi, l'operazione elimina gli spicchi a bassi numeri d'ordine. Se inizialmente invece ha meno di *m* spicchi, l'operazione aggiunge nuovi spicchi ad alti numeri d'ordine; tali nuovi spicchi sono trasparenti. Se inizialmente essa ha esattamente *m* spicchi, l'operazione lascia la ruota inalterata.
- *String comeStringa()*: restituisce una stringa del tipo *<10> –XXX-X-X-* dove le parentesi angolate racchiudono il numero degli spicchi che compongono la ruota, il carattere *'-'* rappresenta uno spicchio trasparente, ed il carattere *'X'* rappresenta uno spicchio opaco.

Realizzare il tipo di dato *Ruota* facendo ricorso ad una rappresentazione a lista (se opportuno, fare uso di una classe ausiliaria *Spicchio* dotata dei metodi necessari).

Soluzione

Ruota.java:

```
1  package cap8.ruota;
2
3  enum Stato {trasparente, opaco}
4
5  class Spicchio {
6    private Stato stato;
7    private Spicchio next;
8    Spicchio getNext() {
9      return next;
10   }
11   void setNext(Spicchio n) {
12     next = n;
13   }
14   Stato getStato() {
15     return stato;
16   }
17   void setStato(Stato s) {
18     stato = s;
19   }
20   void diventaOpaco() {
21     stato = Stato.opaco;
22   }
23   void diventaTrasparente() {
24     stato = Stato.trasparente;
25   }
26   boolean seiOpaco() {
27     return stato == Stato.opaco;
28   }
29   void sovrapponi(Spicchio s) {
30     setStato(
31       ((stato == Stato.trasparente) &&
32        (s.getStato() == Stato.trasparente))
33       ? Stato.trasparente : Stato.opaco);
34   }
35   String comeStringa() {
36     return (stato == Stato.trasparente) ? "-" : "X";
37   }
38 }
```

```
39
40  public class Ruota {
41    private int quanti;
42    private Spicchio ultimo;
43    public Ruota() {
44      ultimo = new Spicchio();
45      ultimo.setNext(ultimo);
46      quanti = 1;
47    }
48    public Ruota(int n) {
49      if (n <= 0)
50        n = 1;
51      ultimo = new Spicchio();
52      Spicchio p = ultimo;
53      for (int i = 1; i < n; i++) {
54        p.setNext(new Spicchio());
55        p = p.getNext();
56      }
57      p.setNext(ultimo);
58      quanti = n;
59    }
60    private void contrai(int n) {
61      Spicchio p = ultimo.getNext();
62      Spicchio q = p.getNext();
63      for (int i = 0; i < n; i++) {
64        q = p.getNext();
65        p = q;
66      }
67      ultimo.setNext(p);
68      quanti -= n;
69    }
70    private void espandi(int n) {
71      Spicchio primo = ultimo.getNext();
72      Spicchio p = ultimo;
73      for (int i = 0; i < n; i++) {
74        Spicchio q = new Spicchio();
75        p.setNext(q);
76        p = q;
77      }
78      ultimo = p;
79      ultimo.setNext(primo);
```

```
80        quanti += n;
81      }
82      public void spicchi(int m) {
83        if (m > 0) {
84          if (quanti > m)
85            contrai(quanti - m);
86          else if (quanti < m)
87            espandi(m - quanti);
88        }
89      }
90      private Spicchio dammiSpicchio(int s) {
91        Spicchio p = ultimo;
92        for (int i = 0; i < s; i++)
93          p = p.getNext();
94        return p;
95      }
96      public void rendiOpaco(int i) {
97        dammiSpicchio(i % quanti).diventaOpaco();
98      }
99      public void rendiTrasparente(int i) {
100       dammiSpicchio(i % quanti).diventaTrasparente();
101     }
102     public boolean seiOpaco(int i) {
103       return dammiSpicchio(i % quanti).seiOpaco();
104     }
105     public void gira(int p) {
106       for (int i = quanti - (p % quanti); i > 0; i--)
107         ultimo = ultimo.getNext();
108     }
109     public void sovrapponi(Ruota t) {
110       if (quanti != t.quanti)
111         return;
112       else {
113         Spicchio p = ultimo;
114         Spicchio q = t.ultimo;
115         for (int i = 0; i < quanti; i++) {
116           p.sovrapponi(q);
117           p = p.getNext();
118           q = q.getNext();
119         }
120       }
```

```
121     }
122     public String comeStringa() {
123       String s = "<" + quanti + "> ";
124       Spicchio q = ultimo.getNext();
125       for (int i = 0; i < quanti; i++) {
126         s += q.comeStringa();
127         q = q.getNext();
128       }
129       return s;
130     }
131   }
```

7.4 IperSfera

Ogni sfera è colorata e ha un numero intero positivo inciso sopra di essa. I possibili colori sono rosso, giallo e verde. Le operazioni che possono essere effettuate su una sfera sono le seguenti:

- *Sfera()*: costruttore di default, che crea una sfera di colore *rosso* e numero 1.
- *Sfera(c, i)*: costruttore che crea una sfera di colore *c* e numero *i*.
- *boolean uguale(Sfera f)*: restituisce *true* se il valore della sfera *f* è uguale al valore della sfera implicita (stesso colore e stesso numero), e *false* altrimenti.
- *Colore qualeColore()*: restituisce il colore della sfera.
- *int qualeNumero()*: restituisce il numero della sfera.
- *String toString()*: la stringa restituita ha la forma *<giallo, 5>*.

Un'ipersfera è in grado di contenere sfere. Le operazioni che possono essere effettuate su un'ipersfera sono le seguenti:

- *IperSfera()*: costruttore di default, che crea un'ipersfera *d*. Inizialmente tale ipersfera è vuota.
- *void aggiungi(Sfera f)*: aggiunge all'ipersfera una sfera avente il colore ed il numero della sfera *f*.
- *void rimuovi(Sfera f)*: estrae la sfera avente il colore ed il numero della sfera *f*, inserita da più tempo nell'ipersfera. Se l'ipersfera non contiene almeno una sfera avente il colore ed il numero della sfera *f*, l'operazione lascia l'ipersfera inalterata.

- *int vecchiaSfera(Colore c)*: restituisce il numero della sfera di colore *c* inserita da più tempo nell'ipersfera. Se l'ipersfera non contiene almeno una sfera di colore *c*, il metodo restituisce 0.

Realizzare il tipo di dati astratti *Sfera*, definito dalle precedenti specifiche. Utilizzare il tipo *Sfera* per realizzare il tipo di dati astratti *IperSfera*. Fare ricorso ad una rappresentazione dell'ipersfera a lista.

Soluzione

Colore.java:

```
1  package cap7.ipersfera;
2
3  public enum Colore {
4    rosso, verde, giallo;
5  }
```

Sfera.java:

```
1  package cap7.ipersfera;
2
3  public class Sfera {
4    private int num;
5    private Colore col;
6
7    public Sfera() {
8      col = Colore.rosso;
9      num = 1;
10   }
11
12   public Sfera(Colore c, int n) {
13     col = c;
14     num = n;
15   }
16
17   public boolean uguale(Sfera o) {
18     return ((o.col == col) && (o.num == num));
19   }
20
21   public Colore qualeColore() {
22     return col;
```

```
23    }
24
25    public int qualeNumero() {
26      return num;
27    }
28
29    public String toString() {
30      return "<" + col + ", " + num + ">";
31    }
32  }
```

IperSfera.java:

```
1   package cap7.ipersfera;
2
3   class Elem {
4     Sfera inf;
5     Elem next;
6
7     Elem(Sfera s) {
8       inf = s;
9     }
10  }
11
12
13  public class IperSfera {
14    private Elem testa;
15
16    public void aggiungi(Sfera s) {
17      if (testa == null) {
18        testa = new Elem(s);
19      } else {
20        Elem aux = testa;
21        while (aux.next != null)
22          aux = aux.next;
23        aux.next = new Elem(s);
24      }
25    }
26
27    public void rimuovi(Sfera f) {
28      if (testa == null) {
29        return;
```

```
30        }
31        Elem aux = testa;
32        if (aux.inf.equals(f)) {
33          testa = testa.next;
34          return;
35        }
36        while (aux.next != null) {
37          if (aux.next.inf.equals(f)) {
38            Elem aux1 = aux.next;
39            aux.next = aux1.next;
40            return;
41          }
42          aux = aux.next;
43        }
44        return;
45      }
46
47      public int vecchiaSfera(Colore c) {
48        Elem aux = testa;
49        while (aux != null) {
50          if (aux.inf.qualeColore() == c) {
51            return aux.inf.qualeNumero();
52          }
53          aux = aux.next;
54        }
55        return 0;
56      }
57
58      public String toString() {
59        String s = "<";
60        if (testa != null) {
61          s += testa.inf;
62          Elem e = testa.next;
63          while (e != null) {
64            s += ", ";
65            s += e.inf;
66            e = e.next;
67          }
68        }
69        s += ">";
70        return s;
```

```
71    }
72  }
```

Menu.java:

```
1   package cap7.ipersfera;
2
3   import IngressoUscita.Console;
4   class Menu {
5     static void stampaMenu() {
6       Console.scriviStringa("1) Aggiungi una sfera");
7       Console.scriviStringa("2) Rimuovi una sfera");
8       Console.scriviStringa("3) Vecchia sfera");
9       Console.scriviStringa("4) Esci");
10    }
11    static Sfera menuSfera() {
12      Colore colore = menuColore();
13      Console.scriviStringa("Inserisci numero");
14      int i = Console.leggiIntero();
15      return new Sfera(colore, i);
16    }
17    static Colore menuColore() {
18      Console.scriviStringa("Inserisci colore (r, v, g)");
19      char c = Console.leggiCarattere();
20      Colore colore;
21      switch (c) {
22      case 'r':
23        colore = Colore.rosso;
24        break;
25      case 'v':
26        colore = Colore.verde;
27        break;
28      case 'g':
29        colore = Colore.giallo;
30        break;
31      default:
32        colore = Colore.rosso;
33      }
34      return colore;
35    }
36    public static void main(String[] args) {
37      IperSfera is = new IperSfera();
```

```
38        Sfera sf;
39        for (;;) {
40          stampaMenu();
41          int i = Console.leggiIntero();
42          switch (i) {
43          case 1:
44            sf = menuSfera();
45            is.aggiungi(sf);
46            break;
47          case 2:
48            sf = menuSfera();
49            is.rimuovi(sf);
50            break;
51          case 3:
52            Colore c = menuColore();
53            Console.scriviIntero(is.vecchiaSfera(c));
54            break;
55          case 4:
56            return;
57          }
58          Console.scriviStringa(is.toString());
59        }
60      }
61    }
```

8. Altre proprietà delle classi

8.1 Acquisti

Un negozio possiede un certo numero di articoli di vario tipo. Ogni tipo di articolo è identificato da un numero e, per ogni tipo, vi sarà un prezzo e un certo numero di pezzi disponibili.

Scrivere una classe *Negozio* che modella il negozio descritto precedentemente. Supporre che vi sia un massimo noto a-priori per il numero di possibili tipi diversi di articoli. Usare il tipo primitivo *double* per esprimere prezzi e quantità di denaro. La classe deve contenere almeno i seguenti metodi:

- *aggiungi*: aggiunge un certo numero di pezzi alla disponibilità di un certo tipo di articolo.
- *fissaPrezzo*: fissa il prezzo di un certo tipo di articoli.
- *compra*: chi invoca il metodo deve poter specificare il tipo di articolo desiderato, il numero di pezzi di quell'articolo che intende acquistare e la quantità di denaro a sua disposizione; il metodo deve restituire al chiamante il numero di pezzi effettivamente acquistati (che possono essere meno di quelli richiesti per via della limitata disponibilità o perchè il denaro messo a disposizione è insufficiente) e l'eventuale resto.

Per ogni metodo, decidere quali devono essere i parametri. Se necessario, introdurre altre classi oltre alla classe *Negozio*. Ogni metodo deve restituire *false* se i parametri ricevuti non hanno senso (per esempio, identificatore non valido, prezzo negativo, etc.) e *true* altrimenti.

Infine, scrivere un programma di test che, ciclicamente, mostra sul video la quantità di denaro a disposizione dell'utente (1000.00€ all'inizio dell'esecuzione) e un menù dal quale è possibile scegliere di eseguire una delle tre azioni precedenti, o uscire.

Soluzione

Pezzi.java:

```
1  package cap8.acquisti;
2
3  public class Pezzi {
4    public int num;
5  }
```

Soldi.java:

```
1  package cap8.acquisti;
2
3  public class Soldi {
4    public double tot;
5  }
```

Negozio.java:

```
1  package cap8.acquisti;
2
3  public class Negozio {
4    public static final int MAXID = 10;
5    private static class Articolo {
6      int disp;
7      double prezzo;
8    }
9    private Articolo[] articoli = new Articolo[MAXID];
10   private boolean idValido(int id) {
11     return (id >= 0) && (id < MAXID) &&
12            (articoli[id] != null);
13   }
14   public boolean aggiungi(int id, int num) {
15     if ((id < 0) || (id > MAXID) || (num <= 0))
16       return false;
17     if (articoli[id] == null)
18       articoli[id] = new Articolo();
19     int num2 = articoli[id].disp + num;
20     articoli[id].disp = num2;
21     return true;
22   }
23   public boolean fissaPrezzo(int id, double p) {
```

```
24      if (!idValido(id) || (p <= 0))
25        return false;
26      articoli[id].prezzo = p;
27      return true;
28    }
29    public boolean compra(int id, Pezzi q, Soldi s) {
30      if (!idValido(id) || (q.num < 0) || (s.tot < 0))
31        return false;
32      int num =
33        (q.num < articoli[id].disp) ? q.num :
34        articoli[id].disp;
35      int num2 = (int) (s.tot / articoli[id].prezzo);
36      q.num = ((num < num2) ? num : num2);
37      articoli[id].disp -= q.num;
38      s.tot -= (q.num * articoli[id].prezzo);
39      return true;
40    }
41  }
```

Test.java:

```
1   package cap8.acquisti;
2
3   import IngressoUscita.Console;
4   class Test {
5     public static void main(String[] args) {
6       Soldi s = new Soldi();
7       Negozio n = new Negozio();
8       s.tot = 1000;
9       for (;;) {
10        Console.scriviStringa("Hai: " + s.tot);
11        Console.scriviStringa("1) aggiungi");
12        Console.scriviStringa("2) fissa prezzo");
13        Console.scriviStringa("3) compra");
14        Console.scriviStringa("4) fine");
15        int i = Console.leggiIntero();
16        switch (i) {
17        case 1:
18          Console.scriviStringa("inserisci id e quantita'");
19          if (n.aggiungi(Console.leggiIntero(),
20                         Console.leggiIntero())) {
21            Console.scriviStringa("OK");
```

```
22          } else {
23            Console.scriviStringa("Errore");
24          }
25          break;
26        case 2:
27          Console.scriviStringa("inserisci id e prezzo");
28          if (n.fissaPrezzo(Console.leggiIntero(),
29                            Console.leggiReale())) {
30            Console.scriviStringa("OK");
31          } else {
32            Console.scriviStringa("Errore");
33          }
34          break;
35        case 3:
36          Console.scriviStringa("inserisci id e quantita'");
37          int id = Console.leggiIntero();
38          Pezzi p = new Pezzi();
39          p.num = Console.leggiIntero();
40          if (n.compra(id, p, s)) {
41            Console.scriviStringa("Hai comprato " + p.num +
42                                  " pezzi");
43          } else {
44            Console.scriviStringa("Errore");
45          }
46          break;
47        case 4:
48          return;
49        default:
50          Console.scriviStringa("Errore");
51          break;
52        }
53      }
54    }
55  }
```

Note

Poiché il testo richiede esplicitamente che tutti i metodi debbano restituire un *boolean*, il metodo *compra*, per restituire le altre informazioni che sono richieste (numero di pezzi effettivamente comprati e resto), deve modificare i propri parametri. Per permettere ciò, è stato necessario introdurre le classi *Pezzi* e *Soldi*.

8.2 Biglietti

Realizzare una classe *Biglietto* che rappresenta dei biglietti per uno spettacolo. Solo un numero limitato (e stabilito a priori) di biglietti *validi* devono poter esistere, in ogni istante. Un biglietto valido può essere reso non più valido, invocando il metodo *lascia()* del biglietto stesso. Scrivere quindi un programma di test, che mostra sul video un menù tramite il quale è possibile prendere un nuovo biglietto (se possibile) o lasciare uno dei biglietti già presi.

Soluzione

Biglietto.java:

```
1  package cap8.biglietti;
2
3  class Biglietto {
4    private static int quanti = 0;
5    private static final int MAX = 10;
6    private boolean valido;
7
8    private Biglietto() {
9      valido = true;
10   }
11
12   public static int disponibili() {
13     return MAX - quanti;
14   }
15
16   public static Biglietto prendi() {
17     if (quanti >= MAX) {
18       return null;
19     }
20     quanti++;
21     return new Biglietto();
22   }
23
24   public String toString() {
25     return "Biglietto " + (valido ? "valido" :
26                            "non valido");
27   }
28
29   public void lascia() {
```

```
30      if (valido) {
31        valido = false;
32        quanti--;
33      }
34    }
35  }
```

Test.java:

```
1   package cap8.biglietti;
2
3   import IngressoUscita.Console;
4   public class Test {
5     public static void main(String[] args) {
6       boolean ancora = true;
7       Biglietto[] biglietti = new Biglietto[100];
8       int ultimo = 0;
9       String scelta;
10      while (ancora && (ultimo < 100)) {
11        Console.scriviStringa("Biglietti disponibili: " +
12                              Biglietto.disponibili());
13        Console.scriviStringa("p) Prendi un biglietto");
14        Console.scriviStringa("l) Lascia un biglietto");
15        Console.scriviStringa("e) Esci");
16        scelta = Console.leggiStringa();
17        switch (scelta.charAt(0)) {
18        case 'p':
19          if ((biglietti[ultimo] = Biglietto.prendi()) !=
20              null) {
21            ultimo++;
22          } else {
23            Console.scriviStringa("Non ci sono biglietti");
24          }
25          break;
26        case 'e':
27          ancora = false;
28          break;
29        case 'l':
30          for (int i = 0; i < ultimo; i++)
31            Console.scriviStringa("" + i + ") lascia " +
32                                  biglietti[i]);
33          int num = Console.leggiIntero();
```

```
34            if ((num < 0) || (num >= ultimo)) {
35              Console.scriviStringa("Scelta non valida");
36            } else {
37              biglietti[num].lascia();
38            }
39            break;
40          default:
41            Console.scriviStringa("Scelta non valida");
42            break;
43          }
44        }
45      }
46    }
```

8.3 Spettacoli

Nella soluzione dell'esercizio 8.2, tutti i biglietti facevano riferimento ad un unico "spettacolo", in quanto il numero di biglietti validi veniva conteggiato da un'unica variabile (statica) della classe *Biglietto*. Modificare la soluzione dell'esercizio in modo che sia possibile creare più spettacoli, specificando, al momento della creazione, il nome dello spettacolo e quanti biglietti validi devono esistere, per quello spettacolo.

Scrivere un programma di test che mostra all'utente un menù tramite il quale è possibile creare un nuovo spettacolo, prendere un biglietto di uno degli spettacoli disponibili, lasciare uno dei biglietti presi precedentemente, uscire dal programma.

Suggerimenti

Definire una classe *Spettacolo*. Può essere utile definire la classe *Biglietto* come una classe *interna* alla classe *Spettacolo*.

Soluzione

Spettacolo.java:

```
1   package cap8.spettacoli;
2
3   class Spettacolo {
4     private int quanti;
5     private int max;
6     private String nome;
```

```
7     Spettacolo(String nome, int max) {
8       this.nome = nome;
9       this.max = max;
10    }
11    public int disponibili() {
12      return max - quanti;
13    }
14    public String toString() {
15      return nome + "(" + disponibili() + "/" + max + ")";
16    }
17    public class Biglietto {
18      private boolean valido;
19      public Biglietto() {
20        if (quanti < max) {
21          quanti++;
22          valido = true;
23        } else
24          valido = false;
25      }
26      public String toString() {
27        return "Biglietto " +
28               (valido ? "valido" : " non valido") + " per " + nome;
29      }
30      public void lascia() {
31        if (valido) {
32          valido = false;
33          quanti--;
34        }
35      }
36    }
37  }
```

Test.java:

```
1   package cap8.spettacoli;
2
3   import IngressoUscita.Console;
4   class Test {
5     private static final int MAX_S = 10;
6     private static final int MAX_B = 100;
7     public static void main(String[] args) {
8       Spettacolo[] s = new Spettacolo[MAX_S];
```

```
9       Spettacolo.Biglietto[] b =
10        new Spettacolo.Biglietto[MAX_B];
11      int num_s = 0;
12      int num_b = 0;
13      int scelta;
14      while ((num_s < MAX_S) && (num_b < MAX_B)) {
15        Console.scriviStringa(
16          "Spettacoli disponibili: " + num_s);
17        Console.scriviStringa("1) Crea un nuovo spettacolo");
18        Console.scriviStringa("2) Prendi un biglietto");
19        Console.scriviStringa("3) Lascia un biglietto");
20        Console.scriviStringa("4) Esci");
21        int op = Console.leggiIntero();
22        if ((op < 1) || (op > 4)) {
23          Console.scriviStringa("Scelta non valida");
24          continue;
25        }
26        if (op == 4) {
27          Console.scriviStringa("Ciao");
28          break;
29        }
30        switch (op) {
31        case 1:
32          Console.scriviStringa("nome?");
33          String nome = Console.leggiStringa();
34          Console.scriviStringa("numero biglietti?");
35          int num = Console.leggiIntero();
36          s[num_s++] = new Spettacolo(nome, num);
37          break;
38        case 2:
39          for (int i = 0; i < num_s; i++)
40            Console.scriviStringa("" + i + ") " + s[i]);
41          Console.scriviStringa("Scegli lo spettacolo");
42          scelta = Console.leggiIntero();
43          if ((scelta < 0) || (scelta > num_s)) {
44            Console.scriviStringa("Scelta non valida");
45            continue;
46          }
47          b[num_b++] = s[scelta].new Biglietto();
48          break;
49        case 3:
```

```
50            for (int i = 0; i < num_b; i++)
51              Console.scriviStringa("" + i + ") " + b[i]);
52            Console.scriviStringa("Scegli il biglietto");
53            scelta = Console.leggiIntero();
54            if ((scelta < 0) || (scelta > num_b)) {
55              Console.scriviStringa("Scelta non valida");
56              continue;
57            }
58            b[scelta].lascia();
59            break;
60          default:
61            break;
62          }
63        }
64      }
65    }
```

Note

Poiché la classe *Biglietto* è interna alla classe *Spettacolo*, ogni oggetto di tipo *Biglietto* mantiene un riferimento implicito ad un oggetto di tipo *Spettacolo*. In questo modo, la variabile *quanti* utilizzata alle linee 20, 21 e 34 della classe *Spettacolo* si riferisce al numero di biglietti per lo specifico spettacolo a cui il biglietto appartiene (linea 4).

8.4 Mia console

Supponiamo che la classe *Console* disponga solo dei metodi *String leggiStringa()* e *void scriviStringa(String s)*, e che in particolare non siano presenti i metodi *int leggiIntero()*, *void scriviIntero(int i)*, *double leggiReale()*, e *void scriviReale()*. Scrivere una classe *MiaConsole* che realizza tali metodi, e in aggiunta anche i metodi *void scriviIntero(Integer i)* e *void scriviReale(Double d)*.

Soluzione

MiaConsole.java:

```
1   package cap8.miaconsole;
2
3   import IngressoUscita.Console;
4   public class MiaConsole {
```

```
5     public static int leggiIntero() {
6       String s = Console.leggiStringa();
7       return Integer.parseInt(s);
8     }
9     public static double leggiReale() {
10      String s = Console.leggiStringa();
11      return Double.parseDouble(s);
12    }
13    public static void scriviIntero(int i) {
14      Console.scriviStringa(String.valueOf(i));
15    }
16    public static void scriviIntero(Integer i) {
17      Console.scriviStringa(i.toString());
18    }
19    public static void scriviReale(double d) {
20      Console.scriviStringa(String.valueOf(d));
21    }
22    public static void scriviReale(Double d) {
23      Console.scriviStringa(d.toString());
24    }
25  }
```

9. Derivazione

9.1 Pila

Una pila è un insieme di dati in cui è possibile effettuare operazioni di inserimento e di estrazione secondo una regola di accesso LIFO (Last In First Out): l'ultimo dato ad essere inserito è il primo ad essere estratto.

Una pila può essere costituita da un array che memorizza i dati nelle sue componenti e da un indice che individua il livello di riempimento della pila (in questo caso la pila ha una dimensione massima pari alla dimensione dell'array).

Su una pila vengono generalmente definite funzioni di inserimento di un nuovo dato, estrazione di un dato, lettura (senza estrazione) del dato in cima alla pila, verifica della condizione pila piena, verifica della condizione pila vuota.

Realizzare il tipo di dati astratti pila di *Object*, definito dalle precedenti specifiche. Inoltre, dotare la classe dei metodi descritti di seguito.

- *Pila(int s)*: costruisce una pila la cui massima grandezza è *s*.
- *boolean equals(Object o)*: restituisce *true* se l'oggetto passato come parametro è una pila uguale a quella su cui viene invocato il metodo, false altrimenti;
- *String toString()*: restituisce una rappresentazione della pila in formato stringa.

Scrivere infine una classe ausiliaria che consente di operare su una pila per inserire ed estrarre stringhe (*String* come tutte le classi è un sottotipo di *Object* e pertanto sue istanze possono essere gestite da una pila di *Object*).

Soluzione

Pila.java:

```
1  package cap9.pila;
2
3  public class Pila {
4    static private final int DEFAULT_SIZE = 10;
5    protected int size;
6    protected int top;
7    protected Object[] v;
8    public Pila() {
9      this(DEFAULT_SIZE);
10   }
11   public Pila(int s) {
12     size = s;
13     top = -1;
14     v = new Object[size];
15   }
16   public boolean isEmpty() {
17     return top == -1;
18   }
19   public boolean isFull() {
20     return top == (size - 1);
21   }
22   public boolean push(Object elem) {
23     if (isFull() || (elem == null))
24       return false;
25     top++;
26     v[top] = elem;
27     return true;
28   }
29   public Object pop() {
30     if (isEmpty())
31       return null;
32     Object result = v[top];
33     top--;
34     return result;
35   }
36   public Object top() {
37     if (isEmpty())
38       return null;
39     return v[top];
40   }
41   public String toString() {
```

```
42      String ls = System.getProperty("line.separator");
43      String s = "---" + ls;
44      if (!isEmpty()) {
45        for (int i = top; i >= 0; i--)
46          s += (v[i] + ls);
47      }
48      s += "---";
49      return s;
50    }
51    public boolean equals(Object o) {
52      if ((o == null) || !(o instanceof Pila))
53        return false;
54      Pila p = (Pila) o;
55      if ((p.top != top) || (p.size != size))
56        return false;
57      for (int i = top; i >= 0; i--)
58        if (!v[i].equals(p.v[i]))
59          return false;
60      return true;
61    }
62  }
```

MenuPila.java:

```
1   package cap9.pila;
2
3   import IngressoUscita.Console;
4   class MenuPila {
5     private static void stampaMenu() {
6       Console.scriviStringa("Scegli:");
7       Console.scriviStringa(
8         "1 Inserisci una stringa nella pila");
9       Console.scriviStringa(
10        "2 Estrai una stringa dalla pila");
11      Console.scriviStringa(
12        "3 Stampa il contenuto della pila");
13      Console.scriviStringa("4 Verifica se la pila e' vuota");
14      Console.scriviStringa("5 Verifica se la pila e' piena");
15      Console.scriviStringa("6 Esci");
16    }
17    public static void main(String[] args) {
18      Pila pila = new Pila();
```

```
19      for (;;) {
20        stampaMenu();
21        int s = Console.leggiIntero();
22        switch (s) {
23        case 1:
24          Console.scriviStringa("Stringa da inserire ?");
25          String x = Console.leggiStringa();
26          if (!pila.push(x))
27            Console.scriviStringa(
28              "Il dato non e' stato inserito (pila piena)");
29          break;
30        case 2:
31          Object o = pila.pop();
32          if (o != null)
33            Console.scriviStringa(
34              "La stringa estratta e' " + o);
35          else
36            Console.scriviStringa(
37              "Non posso estrarre: la pila e' vuota");
38          break;
39        case 3:
40          Console.scriviStringa(pila.toString());
41          break;
42        case 4:
43          Console.scriviStringa(
44            "Pila vuota: " +
45            (pila.isEmpty() ? "vero" : "falso"));
46          break;
47        case 5:
48          Console.scriviStringa(
49            "Pila piena: " +
50            (pila.isFull() ? "vero" : "falso"));
51          break;
52        case 6:
53          System.exit(0);
54        default:
55          Console.scriviStringa("Scelta non valida");
56        }
57      }
58    }
59  }
```

Note

Il separatore di linea viene ricavato dalle proprietà di sistema, tramite il metodo *System.getProperty("line.separator")*, in modo tale che corrisponda al separatore in uso sulla macchina che esegue il programma.

9.2 Pila ordinabile

Scrivere una interfaccia *Comparabile* avente il seguente metodo:
int comparaCon(Comparabile c)
Le classi che implementano l'interfaccia *Comparabile* devono fornire una realizzazione del metodo tale da esprimere una relazione di ordinamento tra due oggetti: se il valore restituito è minore di zero, l'oggetto implicito precede quello passato come argomento, se è positivo, l'oggetto implicito segue quello passato come argomento, e se infine il valore restituito è zero i due oggetti sono equivalenti.

Definire, a questo punto, una classe *MioIntero* che implementa l'interfaccia *Comparabile*.

Realizzare infine una sottoclasse della classe *Pila* che aggiunga a quest'ultima un metodo *void ordina()* che, quando viene invocato, ordina i dati contenuti nella pila in modo crescente (sulla cima della pila c'è l'elemento più grande). È necessario anche ridefinire il metodo *inserisci()* della classe *Pila* in modo tale che sia possibile inserire solo oggetti comparabili.

Soluzione

Comparabile.java:

```
1  package cap9.pilaordinabile;
2
3  public interface Comparabile {
4    int comparaCon(Comparabile c);
5  }
```

MioIntero.java:

```
1  package cap9.pilaordinabile;
2
3  public class MioIntero implements Comparabile {
4    private int value;
5
6    public MioIntero(int v) {
7      value = v;
```

```
8     }
9
10    public int comparaCon(Comparabile c) {
11      MioIntero m = (MioIntero) c;
12      return value - m.value;
13    }
14
15    public String toString() {
16      return String.valueOf(value);
17    }
18  }
```

PilaOrdinata.java:

```
1   package cap9.pilaordinabile;
2
3   import cap9.pila.Pila;
4   public class PilaOrdinata extends Pila {
5     public void ordina() {
6       Comparabile temp;
7       boolean ordinato;
8       if (isEmpty())
9         return;
10      for (int fine = top; fine > 0; fine--) {
11        ordinato = true;
12        for (int i = 0; i < fine; i++)
13          if (
14            ((Comparabile) v[i]).comparaCon(
15              (Comparabile) v[i + 1]) > 0) {
16            temp = (Comparabile) v[i];
17            v[i] = v[i + 1];
18            v[i + 1] = temp;
19            ordinato = false;
20          }
21        if (ordinato)
22          return;
23      }
24    }
25    public boolean push(Object o) {
26      if (o instanceof Comparabile)
27        return super.push(o);
28      return false;
```

```
29    }
30  }
```

MenuPilaOrdinata.java:

```
1   package cap9.pilaordinabile;
2
3   import IngressoUscita.Console;
4   class MenuPilaOrdinata {
5     private static void stampaMenu() {
6       Console.scriviStringa("Scegli:");
7       Console.scriviStringa(
8         "1 Inserisci un MioIntero nella pila");
9       Console.scriviStringa(
10        "2 Estrai un MioIntero dalla pila");
11      Console.scriviStringa(
12        "3 Stampa il contenuto della pila");
13      Console.scriviStringa("4 Verifica se la pila e' vuota");
14      Console.scriviStringa("5 Verifica se la pila e' piena");
15      Console.scriviStringa("6 Ordina la pila");
16      Console.scriviStringa("7 Esci");
17    }
18    public static void main(String[] args) {
19      PilaOrdinata pila = new PilaOrdinata();
20      for (;;) {
21        stampaMenu();
22        int s = Console.leggiIntero();
23        switch (s) {
24        case 1:
25          Console.scriviStringa("Valore da inserire? ");
26          int x = Console.leggiIntero();
27          if (!pila.push(new MioIntero(x)))
28            Console.scriviStringa(
29              "Il dato " +
30              "non e' stato inserito (pila piena o oggetto nullo)");
31          break;
32        case 2:
33          Object o = pila.pop();
34          if (o != null)
35            Console.scriviStringa(
36              "Il dato estratto e' " + o);
37          else
```

```
38              Console.scriviStringa(
39                "Non posso estrarre:" + "la pila e' vuota");
40            break;
41          case 3:
42            Console.scriviStringa(pila.toString());
43            break;
44          case 4:
45            Console.scriviStringa(
46              "Pila vuota: " +
47              (pila.isEmpty() ? "vero" : "falso"));
48            break;
49          case 5:
50            Console.scriviStringa(
51              "Pila piena: " +
52              (pila.isFull() ? "vero" : "falso"));
53            break;
54          case 6:
55            pila.ordina();
56            break;
57          case 7:
58            System.exit(0);
59          default:
60            Console.scriviStringa("Scelta non valida");
61          }
62        }
63      }
64    }
```

9.3 Pila iterabile

Un *iteratore* è un oggetto che permette di scandire linearmente gli elementi di una struttura dati. In Java, possiamo introdurre la seguente interfaccia, che deve essere realizzata da tutti gli iteratori [1]:

Iterator.java:

```
1   package cap9.pilaiterabile;
2
```

[1] Le librerie del linguaggio includono una interfaccia *Iterator* (package *java.util*) implementata da numerose classi di sistema. Tale interfaccia, oltre ai metodi *next()* e *hasNext()* ne include un altro, *remove()*, che consente di rimuovere elementi. Qui, per motivi di semplicità, abbiamo ritenuto opportuno utilizzarne una versione ridotta.

```
3   public interface Iterator {
4     Object next();
5
6     boolean hasNext();
7   }
```

Dove:

- *boolean hasNext()*: restituisce *true* se ci sono ancora elementi, *false* altrimenti.
- *Object next()*: restituisce il riferimento al successivo elemento nell'iterazione.

Si realizzi la classe *PilaIterabile*, derivata dalla classe *Pila* dell'esercizio 9.1, che permette la creazione di oggetti *Iterator* per scandire tutti gli elementi della pila. Si realizzino due tipi di iteratori: uno (*IteratoreIndietro*) per scandire gli elementi dall'ultimo inserito al primo, l'altro (*IteratoreAvanti*) per scandirli dal primo inserito all'ultimo.

Suggerimenti

Si realizzino le classi *IteratoreAvanti* e *IteratoreIndietro* come classi interne della classe *PilaIterabile*

Soluzione

PilaIterabile.java:

```
1   package cap9.pilaiterabile;
2
3   import cap9.pila.Pila;
4   public class PilaIterabile extends Pila {
5     public PilaIterabile(int max) {
6       super(max);
7     }
8     public PilaIterabile() {
9       super();
10    }
11    public Iterator iteratoreAvanti() {
12      return new IteratoreAvanti();
13    }
14    public Iterator iteratoreIndietro() {
15      return new IteratoreIndietro();
```

```
16      }
17      public class IteratoreIndietro implements Iterator {
18        private int pos = top;
19        public boolean hasNext() {
20          if (pos >= 0) {
21            return true;
22          }
23          return false;
24        }
25        public Object next() {
26          if (pos >= 0) {
27            return v[pos--];
28          } else {
29            return null;
30          }
31        }
32      }
33      public class IteratoreAvanti implements Iterator {
34        private int pos = 0;
35
36        public boolean hasNext() {
37          if (pos <= top) {
38            return true;
39          }
40          return false;
41        }
42
43        public Object next() {
44          if (pos <= top) {
45            return v[pos++];
46          } else {
47            return null;
48          }
49        }
50      }
51    }
```

Note

Un iteratore può essere ottenuto richiamando i metodi *Iterator iteratoreAvanti()* o *Iterator iteratoreIndietro()*, oppure applicando l'operatore *new* ad una istan-

za di *PilaIterabile*. Il seguente frammento di codice mostra i due diversi modi supponendo che la pila contenga delle stringhe.

```
...
Iterator ii1 = pila.iteratoreAvanti();
while(ii1.hasNext()) {
  String s1 = (String) ii1.next();
  Console.scriviStringa(s1);
}
Console.scriviStringa("indietro:");
Iterator ii2 = pila.new IteratoreIndietro();
while(ii2.hasNext()) {
  String s2 = (String) ii2.next();
  Console.scriviStringa(s2);
}
...
```

9.4 Menù

Un menù mostra sullo schermo una serie di opzioni numerate, chiede all'utente di scegliere una delle opzioni disponibili, dopodiché il programma principale eseguirà l'azione associata all'opzione scelta. Allo scopo di rendere riutilizzabile il codice associato ad un qualunque menù, si realizzino le seguenti classi e interfacce.

L'interfaccia *Opzione*, contenente:

- *String getDescr()*: metodo che restituisce la stringa che descrive l'opzione nel menù.
- *boolean fai()*: metodo che esegue l'azione associata all'opzione e restituisce *false* se, per effetto dell'azione, deve causare l'uscita dal menù.

La classe *Menu*, contenente:

- *Menu(int max)*: costruttore che crea un *Menù* che potrà contenere al massimo *max* opzioni.
- *boolean aggiungi(Opzione o)*: metodo che aggiunge *o* alle opzioni del *Menù* (restituisce *false* se non è stato possibile aggiungere l'opzione).
- *Opzione scegli()*: metodo che mostra il *Menù* sul video, con le opzioni opportunamente numerate, chiede di inserire il numero dell'opzione prescelta e lo legge da tastiera; quindi, controlla che il numero inserito sia valido: in caso contrario, chiede di reinserire la scelta. Se il numero è valido, restituisce l'oggetto *Opzione* corrispondente.

Infine, riscrivere la classe *Test* dell'esercizio 8.2 in modo che il menù principale sia realizzato usando la classe *Menu* appena definita.

Soluzione

Menu.java:

```
1  package cap9.menu;
2
3  import IngressoUscita.Console;
4  interface Opzione {
5    String getDescr();
6    boolean fai();
7  }
8  class Menu {
9    private Opzione[] opzioni;
10   private int quante;
11   public Menu(int max) {
12     opzioni = new Opzione[max];
13     quante = 0;
14   }
15   public boolean aggiungi(Opzione o) {
16     if (quante < opzioni.length) {
17       opzioni[quante++] = o;
18       return true;
19     } else
20       return false;
21   }
22   public Opzione scegli() {
23     int scelta;
24     for (int i = 1; i <= quante; i++)
25       Console.scriviStringa(
26         i + ") " + opzioni[i - 1].getDescr());
27     for (;;) {
28       Console.scriviStringa("Scegli");
29       String input = Console.leggiStringa();
30       scelta = Integer.parseInt(input);
31       if ((scelta >= 1) && (scelta <= quante))
32         break;
33       Console.scriviStringa("Scelta non valida, ripetere");
34     }
35     return opzioni[scelta - 1];
```

```
36      }
37    }
```

Test.java:

```
1   package cap9.menu;
2
3   import IngressoUscita.Console;
4   abstract class OpzioneGen implements Opzione {
5     protected String descr;
6     public OpzioneGen(String descr) {
7       this.descr = descr;
8     }
9     public String getDescr() {
10      return descr;
11    }
12  }
13  class OpzioniBiglietti {
14    private static Biglietto[] biglietti = new
15    Biglietto[100];
16    private static int ultimo = 0;
17    public static class PrendiBiglietto extends
18      OpzioneGen {
19      public PrendiBiglietto() {
20        super("Prendi un biglietto");
21      }
22      public boolean fai() {
23        if (ultimo >= biglietti.length) {
24          return false;
25        }
26        if ((biglietti[ultimo] = Biglietto.prendi()) !=
27            null) {
28          ultimo++;
29        } else {
30          Console.scriviStringa("Non ci sono biglietti");
31        }
32        return true;
33      }
34    }
35    public static class LasciaBiglietto extends
36      OpzioneGen {
37      public LasciaBiglietto() {
```

```
38        super("Lascia un biglietto");
39      }
40
41      public boolean fai() {
42        for (int i = 0; i < ultimo; i++)
43          Console.scriviStringa("" + i + ") lascia " +
44                                biglietti[i]);
45        int num = Console.leggiIntero();
46        if ((num < 0) || (num >= ultimo)) {
47          Console.scriviStringa("Scelta non valida");
48        } else {
49          biglietti[num].lascia();
50        }
51        return true;
52      }
53    }
54  }
55
56
57  class Esci extends OpzioneGen {
58    public Esci() {
59      super("esci");
60    }
61
62    public boolean fai() {
63      return false;
64    }
65  }
66
67
68  public class Test {
69    public static void main(String[] args) {
70      Menu m = new Menu(3);
71      m.aggiungi(new OpzioniBiglietti.PrendiBiglietto());
72      m.aggiungi(new OpzioniBiglietti.LasciaBiglietto());
73      m.aggiungi(new Esci());
74      Opzione o;
75      do {
76        o = m.scegli();
77      } while (o.fai());
78    }
```

```
79  }
```

Note

La classe *OpzioneGen* è stata definita per raggruppare il codice relativo alla gestione della stringa di descrizione, che, altrimenti, sarebbe stato ripetuto in tutte le classi che realizzano le opzioni.

Notare che, poiché i metodi *fai()* delle opzioni *PrendiBiglietto* e *LasciaBiglietto* (e solo loro) devono accedere alla stessa struttura dati, composta dalla variabili *biglietti* e *ultimo*, si è preferito raggruppare le due classi e la struttura dati nella classe *OpzioniBiglietti*.

9.5 Porte Magiche

Vogliamo estendere il gioco presentato nell'esercizio 6.11 nel seguente modo. Vogliamo introdurre tre tipi diversi di porte. Un primo tipo può essere attraversato soltanto in una direzione (per esempio, soltanto andando dalla prima alla seconda delle stanze specificate al momento della creazione della porta). Un secondo tipo può essere attraversato solo un certo numero di volte (il valore massimo è specificato alla creazione della porta). Infine, un terzo tipo di porta è collegato a tre stanze invece che a due: venendo da una stanza si finisce in una delle altre due, a caso.

Definire le classi *PortaUnidirezionale*, *PortaConLimite* e *PortaCasuale* che estendono *Porta* e implementano i tipi di porte descritti sopra. Modificare anche la classe *Test* dell'esercizio 6.11 in modo che la mappa del labirinto contenga almeno una porta di ogni nuovo tipo.

Soluzione

PortaUnidirezionale.java:

```
1   package cap9.portemagiche;
2
3   import IngressoUscita.Console;
4   import cap6.labirinto.*;
5   public class PortaUnidirezionale extends Porta {
6     public PortaUnidirezionale(
7       Stanza stanza1, Stanza stanza2) {
8       super(stanza1, stanza2);
9     }
10    public Risultato attraversa(Giocatore giocatore) {
11      Risultato r;
```

```
12      if (giocatore.locazione() == stanza1) {
13        r = super.attraversa(giocatore);
14        if (r.isOk())
15          r = new Risultato(
16            true, "La porta si richiude alle tue spalle.");
17      } else
18        r = new Risultato(false, "La porta non si apre.");
19      return r;
20    }
21  }
```

PortaConLimite.java:

```
1   package cap9.portemagiche;
2
3   import IngressoUscita.Console;
4   import cap6.labirinto.*;
5   public class PortaConLimite extends Porta {
6     private int rimanenti;
7     public PortaConLimite(
8       Stanza stanza1, Stanza stanza2,
9       int max_attraversamenti) {
10      super(stanza1, stanza2);
11      rimanenti = max_attraversamenti;
12    }
13    public Risultato attraversa(Giocatore giocatore) {
14      if (rimanenti <= 0)
15        return new Risultato(false, "La porta non si apre.");
16      Risultato r = super.attraversa(giocatore);
17      if (r.isOk()) {
18        r = new Risultato(
19          true,
20          "Senti qualcosa ticchettare " + rimanenti +
21          ((rimanenti == 1) ? " volta." : " volte."));
22        rimanenti--;
23      }
24      return r;
25    }
26  }
```

PortaCasuale.java:

```
1   package cap9.portemagiche;
```

```
2
3   import IngressoUscita.Console;
4   import cap6.labirinto.*;
5   class PortaCasuale extends Porta {
6     private static final int NSTANZE = 3;
7     private Stanza[] stanze = new Stanza[NSTANZE];
8     public PortaCasuale(Stanza stanza1, Stanza stanza2,
9                         Stanza stanza3) {
10      super(stanza1, stanza2);
11      stanze[0] = stanza1;
12      stanze[1] = stanza2;
13      stanze[2] = stanza3;
14    }
15    public Risultato attraversa(Giocatore giocatore) {
16      int i = 0;
17      for (; i < NSTANZE; i++)
18        if (giocatore.locazione() == stanze[i]) {
19          break;
20        }
21      if (i == NSTANZE) {
22        return new Risultato(false, "Errore nella mappa.");
23      }
24      int j = (int) (Math.random() * (NSTANZE - 1)) + 1;
25      giocatore.muovi(stanze[(i + j) % NSTANZE]);
26      return new Risultato(true,
27                           "Nell'attraversare la porta avverti un lieve giramento di testa
28    }
29  }
```

Test.java:

```
1   package cap9.portemagiche;
2
3   import IngressoUscita.Console;
4   import cap6.labirinto.*;
5   class Test {
6     public static void main(String[] args) {
7       Stanza s1 = new
8       Stanza("Stai pilotando il tuo monoposto " +
9              "sopra la desolata regione scozzese delle Highlands");
10      Stanza s2 = new Stanza("Ti trovi in un campo aperto,"
11                             +
```

```
12                          " a ovest di una casa bianca," +
13                          " con la porta principale sbarrata");
14      Stanza s3 = new
15      Stanza("Sei in un labirinto di tortuosi," +
16             " piccoli passaggi, tutti simili");
17      Stanza s4 = new
18      Stanza("Sei in un labirinto di tortuosi," +
19             " piccoli passaggi, tutti diversi");
20      Stanza s5 = new Stanza("Sei nella stanza del tesoro");
21      Porta p1 = new PortaUnidirezionale(s1, s2);
22      s1.collega(Stanza.Direzioni.EST, p1);
23      s2.collega(Stanza.Direzioni.OVEST, p1);
24      Porta p2 = new PortaConLimite(s2, s3, 5);
25      s2.collega(Stanza.Direzioni.EST, p2);
26      s3.collega(Stanza.Direzioni.OVEST, p2);
27      Porta p3 = new PortaCasuale(s3, s4, s5);
28      s3.collega(Stanza.Direzioni.EST, p3);
29      s4.collega(Stanza.Direzioni.SUD, p3);
30      s5.collega(Stanza.Direzioni.NORD, p3);
31      Giocatore giocatore = new Giocatore("Zak");
32      giocatore.muovi(s1);
33      boolean ancora = true;
34      while (ancora) {
35        Stanza s = giocatore.locazione();
36        Console.scriviStringa(s.descrivi());
37        Stanza.Direzioni[] dir = s.direzioni();
38        if (dir.length > 0) {
39          Console.scriviStringa("Puoi andare a:");
40          for (int i = 0; i < dir.length; i++)
41            Console.scriviStringa(dir[i].toString());
42        }
43        Console.scriviStringa("Dove vuoi andare?");
44        String in = Console.leggiStringa();
45        if (in.equals("fine")) {
46          ancora = false;
47        } else if (in.equals("nord") || in.equals("sud")
48                   || in.equals("est") ||
49                  in.equals("ovest")) {
50          Stanza.Direzioni d = Stanza.Direzioni.valueOf(
51                                  in.toUpperCase());
52          Risultato r = s.vai(giocatore, d);
```

```
53            Console.scriviStringa(r.getMessage());
54          } else {
55            Console.scriviStringa("Non capisco la parola " + in);
56          }
57        }
58      }
59    }
```

9.6 Finestre

Nella costruzione di una interfaccia grafica, gli elementi che compongono l'interfaccia vengono normalmente modellati come finestre rettangolari. Alcune di queste finestre sono visibili (bottoni, aree di testo etc.), mentre altre, strutturali, servono a raggruppare le altre finestre e a disporle sullo schermo (per esempio, allineate orizzontalmente o in colonna). Supponiamo di definire soltanto i seguenti tipi di finestra e di essere interessati esclusivamente al calcolo delle loro dimensioni:

- *Bottone*: una finestra visibile, che ha una altezza e una larghezza decisi al momento della creazione.
- *Linea*: una finestra strutturale che contiene altre finestre, disposte orizzontalmente. L'altezza di una *Linea* è il massimo tra le altezze di tutte le finestre contenute, mentre la sua lunghezza è la somma di tutte le lunghezze delle finestre contenute.
- *Colonna*: una finestra strutturale che contiene altre finestre, disposte verticalmente. L'altezza di una *Colonna* è la somma di tutte le altezze delle finestre contenute, mentre la sua lunghezza è il massimo tra le lunghezze di tutte le finestre contenute.

Sia ad un oggetto di tipo *Linea* che ad un oggetto di tipo *Colonna* è possibile aggiungere una nuova finestra in coda a quelle già contenute, oppure estrarre l'ultima finestra inserita. Realizzare le classi che definiscono i tipi di finestra appena descritti, facendo uso delle seguenti definizioni:

Finestre.java:

```
1  package cap9.finestre;
2
3  class Rettangolo {
4    public int larghezza;
5    public int altezza;
6
```

```
7     public Rettangolo(int larghezza, int altezza) {
8       this.larghezza = larghezza;
9       this.altezza = altezza;
10    }
11  }
12
13
14  interface Finestra {
15    Rettangolo dimensioni();
16  }
```

Soluzione

Bottone.java:

```
1   package cap9.finestre;
2
3   class Bottone implements Finestra {
4     protected Rettangolo dim;
5
6     public Bottone(int w, int h) {
7       dim = new Rettangolo(w, h);
8     }
9
10    public Rettangolo dimensioni() {
11      return new Rettangolo(dim.larghezza, dim.altezza);
12    }
13  }
```

Linea.java:

```
1   package cap9.finestre;
2
3   import cap9.pilaiterabile.*;
4   class Linea implements Finestra {
5     protected PilaIterabile finestre;
6     public Linea(int max) {
7       finestre = new PilaIterabile(max);
8     }
9     public void aggiungi(Finestra f) {
10      finestre.push(f);
11    }
```

```
12    public Finestra estrai() {
13      return (Finestra) finestre.pop();
14    }
15    public Rettangolo dimensioni() {
16      Iterator i = finestre.new IteratoreAvanti();
17      Rettangolo tmp = new Rettangolo(0, 0);
18      while (i.hasNext()) {
19        Finestra f = (Finestra) i.next();
20        Rettangolo r = f.dimensioni();
21        if (r.altezza > tmp.altezza) {
22          tmp.altezza = r.altezza;
23        }
24        tmp.larghezza += r.larghezza;
25      }
26      return tmp;
27    }
28  }
```

Colonna.java:

```
1   package cap9.finestre;
2
3   import cap9.pilaiterabile.*;
4   class Colonna implements Finestra {
5     protected PilaIterabile finestre;
6     public Colonna(int max) {
7       finestre = new PilaIterabile(max);
8     }
9     public void aggiungi(Finestra f) {
10      finestre.push(f);
11    }
12    public Finestra estrai() {
13      return (Finestra) finestre.pop();
14    }
15    public Rettangolo dimensioni() {
16      Iterator i = finestre.new IteratoreAvanti();
17      Rettangolo tmp = new Rettangolo(0, 0);
18      while (i.hasNext()) {
19        Finestra f = (Finestra) i.next();
20        Rettangolo r = f.dimensioni();
21        if (r.larghezza > tmp.larghezza) {
22          tmp.larghezza = r.larghezza;
```

```
23          }
24          tmp.altezza += r.altezza;
25        }
26        return tmp;
27      }
28    }
```

9.7 Finestre 2

Le classi *Linea* e *Colonna* nella soluzione dell'esercizio 9.6 contengono molto codice simile. Creare una nuova classe *Collezione* che contenga, il più possibile, il codice comune alle due classi e ridefinire *Linea* e *Colonna* in modo che estendano la classe *Collezione*, ereditando il codice comune e specificando solo le parti in cui differiscono (vale a dire, nel modo in cui l'altezza e la larghezza totale vanno calcolate).

Soluzione

Collezione.java:

```
1   package cap9.finestre2;
2
3   import cap9.pilaiterabile.*;
4   abstract class Collezione implements Finestra {
5     protected PilaIterabile finestre;
6     public Collezione(int max) {
7       finestre = new PilaIterabile(max);
8     }
9     public void aggiungi(Finestra f) {
10      finestre.push(f);
11    }
12    public Finestra estrai() {
13      return (Finestra) finestre.pop();
14    }
15    abstract protected void aggiorna(
16      Rettangolo r1, Rettangolo r2);
17    public Rettangolo dimensioni() {
18      Iterator i = finestre.new IteratoreAvanti();
19      Rettangolo tmp = new Rettangolo(0, 0);
20      while (i.hasNext()) {
21        Finestra f = (Finestra) i.next();
```

```
22          aggiorna(tmp, f.dimensioni());
23        }
24        return tmp;
25      }
26    }
```

Linea.java:

```
1   package cap9.finestre2;
2
3   class Linea extends Collezione {
4     public Linea(int max) {
5       super(max);
6     }
7
8     protected void aggiorna(Rettangolo r1,
9                             Rettangolo r2) {
10      if (r2.altezza > r1.altezza) {
11        r1.altezza = r2.altezza;
12      }
13      r1.larghezza += r2.larghezza;
14    }
15  }
```

Colonna.java:

```
1   package cap9.finestre2;
2
3   class Colonna extends Collezione {
4     public Colonna(int max) {
5       super(max);
6     }
7
8     protected void aggiorna(Rettangolo r1,
9                             Rettangolo r2) {
10      if (r2.larghezza > r1.larghezza) {
11        r1.larghezza = r2.larghezza;
12      }
13      r1.altezza += r2.altezza;
14    }
15  }
```

9.8 Prendere o lasciare

Vogliamo modificare il programma costruito nell'esercizio 6.11, in modo che ogni stanza del labirinto possa contenere oggetti. Il giocatore può prendere alcuni degli oggetti, portarli con se e, eventualmente, lasciarli in un'altra stanza. A questo scopo, definiamo le seguenti interfacce:

Oggetto.java:

```
1  package cap9.prenderelasciare;
2
3  public interface Oggetto {
4    String nome();
5
6    Contenitore luogo();
7  }
```

Contenitore.java:

```
1  package cap9.prenderelasciare;
2
3  import cap6.labirinto.Risultato;
4  public interface Contenitore {
5    Oggetto[] elenco();
6    Risultato aggiungi(Oggetto o);
7    Risultato rimuovi(Oggetto o);
8  }
```

Mobile.java:

```
1  package cap9.prenderelasciare;
2
3  import cap6.labirinto.Risultato;
4  public interface Mobile extends Oggetto {
5    Risultato sposta(Contenitore sorg, Contenitore dest);
6  }
```

L'interfaccia *Oggetto* contiene il metodo *nome()*, che restituisce il nome dell'oggetto e *luogo()*, che restituisce un riferimento al luogo in cui l'oggetto è collocato.

L'interfaccia *Contenitore* modella un contenitore di oggetti. L'interfaccia possiede i metodi: *elenco()*, che restituisce un array di riferimenti a tutti e soli gli oggetti contenuti; *aggiungi()* e *rimuovi()*, che aggiungono/rimuovono un oggetto al contenitore, restituendo un oggetto *Risultato* che contiene *true* se l'operazione ha avuto successo, e *false* altrimenti.

L'interfaccia *Mobile* modella un oggetto che può essere spostato da un contenitore ad un altro. Possiede l'unico metodo *sposta()*, che rimuove l'oggetto dal contenitore *sorg* e lo aggiunge al contenitore *dest*. Se *sorg* è diverso dal contenitore in cui l'oggetto è attualmente contenuto, il metodo deve restituire un oggetto *Risultato* con stato *false* e non deve spostare l'oggetto.

In questo modello, ogni stanza del labirinto è un *Contenitore*, mentre il giocatore stesso è sia un *Contenitore* (può trasportare oggetti) che un *Mobile* (può spostarsi da una stanza ad un'altra). Modificare quindi la soluzione dell'esercizio 6.11 in modo che la classe *Stanza*, oltre alle funzioni richieste dall'esercizio 6.11, implementi anche l'interfaccia *Contenitore*; la classe *Giocatore* implementi le interfacce *Contenitore* e *Mobile*.

Scrivere quindi un programma di test, simile a quello dell'esercizio 6.11, ma che crei anche qualche oggetto, nella descrizione della stanza mostri anche gli oggetti contenuti (compreso il giocatore stesso) e interpreti anche i seguenti comandi:

- "`prendi` *nome oggetto*": il giocatore prende un oggetto, specificandone il nome; l'oggetto deve essere presente nella stanza in cui si trova il giocatore, deve essere prendibile (deve implementare l'interfaccia *Mobile*) e non deve essere già in possesso del giocatore.
- "`lascia` *nome oggetto*": il giocatore lascia un oggetto nella stanza, specificandone il nome (l'oggetto deve essere in possesso del giocatore).
- "`inventario`": mostra l'elenco degli oggetti posseduti dal giocatore.

Soluzione

ContenitoreImpl.java:

```
1  package cap9.prenderelasciare;
2
3  import cap6.labirinto.Risultato;
4  class ContenitoreImpl implements Contenitore {
5    protected final static int MAX_OGGETTI = 10;
6    protected Oggetto[] oggetti;
7    protected int num_oggetti;
8    public ContenitoreImpl() {
9      oggetti = new Oggetto[MAX_OGGETTI];
10     num_oggetti = 0;
11   }
12   public Risultato aggiungi(Oggetto o) {
13     for (int i = 0; i < MAX_OGGETTI; i++) {
14       if (oggetti[i] == null) {
```

```
15          oggetti[i] = o;
16          num_oggetti++;
17          return new Risultato();
18        }
19      }
20      return new Risultato(false, "Non c'e' piu' posto.");
21    }
22    public Risultato rimuovi(Oggetto o) {
23      for (int i = 0; i < MAX_OGGETTI; i++) {
24        if (oggetti[i] == o) {
25          oggetti[i] = null;
26          num_oggetti--;
27          return new Risultato();
28        }
29      }
30      return new Risultato(false, "Non vedo " + o.nome());
31    }
32    public Oggetto[] elenco() {
33      Oggetto[] tmp = new Oggetto[num_oggetti];
34      for (int i = 0, j = 0; i < MAX_OGGETTI; i++) {
35        if (oggetti[i] != null) {
36          tmp[j++] = oggetti[i];
37        }
38      }
39      return tmp;
40    }
41  }
```

MobileImpl.java:

```
1   package cap9.prenderelasciare;
2
3   import cap6.labirinto.Risultato;
4   public class MobileImpl implements Mobile {
5     protected Contenitore luogo;
6     protected String nome;
7     public MobileImpl(String n, Contenitore l) {
8       nome = n;
9       if (l != null) {
10        l.aggiungi(this);
11        luogo = l;
12      }
```

```
13    }
14    public String nome() {
15      return nome;
16    }
17    public Contenitore luogo() {
18      return luogo;
19    }
20    public Risultato sposta(Contenitore sorg,
21                            Contenitore dest) {
22      if (sorg != luogo) {
23        return new Risultato(false,
24                             nome + " si e' spostato.");
25      }
26      Risultato r;
27      if (luogo != null) {
28        r = luogo.rimuovi(this);
29        if (!r.isOk()) {
30          return r;
31        }
32      }
33      r = dest.aggiungi(this);
34      if (!r.isOk()) {
35        luogo.aggiungi(this);
36        return r;
37      }
38      luogo = dest;
39      return new Risultato();
40    }
41  }
```

Stanza.java:

```
1   package cap9.prenderelasciare;
2
3   import cap6.labirinto.Risultato;
4   public class Stanza extends ContenitoreImpl {
5     public enum Direzioni {
6       NORD, SUD, EST, OVEST;
7     }
8     protected String descr;
9     protected Porta[] porte = new Porta[4];
10    public Stanza(String descr) {
```

```
11      this.descr = descr;
12    }
13    public void collega(Direzioni dir, Porta porta) {
14      porte[dir.ordinal()] = porta;
15    }
16    public Risultato vai(Giocatore giocatore,
17                         Direzioni dir) {
18      if (porte[dir.ordinal()] == null)
19        return new Risultato(
20                  false, "Non puoi andare in quella direzione");
21      else
22        return porte[dir.ordinal()].attraversa(giocatore);
23    }
24    public String descrivi() {
25      return descr;
26    }
27    public Direzioni[] direzioni() {
28      int num = 0;
29      for (Direzioni d : Direzioni.values())
30        if (porte[d.ordinal()] != null)
31          num++;
32      Direzioni[] dir = new Direzioni[num];
33      int i = 0;
34      for (Direzioni d : Direzioni.values())
35        if (porte[d.ordinal()] != null) {
36          dir[i] = d;
37          i++;
38        }
39      return dir;
40    }
41  }
```

Giocatore.java:

```
1   package cap9.prenderelasciare;
2
3   import cap6.labirinto.Risultato;
4   public class Giocatore extends ContenitoreImpl
5     implements
6     Mobile {
7     protected Contenitore luogo;
8     protected String nome;
```

```
 9     public Giocatore(String n, Contenitore l) {
10       nome = n;
11       if (l != null) {
12         l.aggiungi(this);
13         luogo = l;
14       }
15     }
16     public String nome() {
17       return nome;
18     }
19     public Contenitore luogo() {
20       return luogo;
21     }
22     public Risultato sposta(Contenitore sorg,
23                             Contenitore dest) {
24       if (sorg != luogo) {
25         return new Risultato(false,
26                              nome + " si e' spostato.");
27       }
28       Risultato r;
29       if (luogo != null) {
30         r = luogo.rimuovi(this);
31         if (!r.isOk()) {
32           return r;
33         }
34       }
35       r = dest.aggiungi(this);
36       if (!r.isOk()) {
37         luogo.aggiungi(this);
38         return r;
39       }
40       luogo = dest;
41       return new Risultato();
42     }
43   }
```

Porta.java:

```
1  package cap9.prenderelasciare;
2
3  import cap6.labirinto.Risultato;
4  public class Porta {
```

```
5     protected Stanza stanza1;
6     protected Stanza stanza2;
7     public Porta(Stanza stanza1, Stanza stanza2) {
8       this.stanza1 = stanza1;
9       this.stanza2 = stanza2;
10    }
11    public Risultato attraversa(Giocatore giocatore) {
12      Stanza stanza = (Stanza) giocatore.luogo();
13      if (stanza == stanza1) {
14        return giocatore.sposta(stanza1, stanza2);
15      } else if (stanza == stanza2) {
16        return giocatore.sposta(stanza2, stanza1);
17      } else {
18        return new Risultato(false, "Errore nella mappa?");
19      }
20    }
21  }
```

Test.java:

```
1   package cap9.prenderelasciare;
2
3   import IngressoUscita.Console;
4   import cap6.labirinto.Risultato;
5   class Test {
6     public static void elenca(String msg, Oggetto[] o) {
7       Console.scriviStringa(msg);
8       if (o.length == 0) {
9         Console.scriviStringa("Niente.");
10        return;
11      }
12      for (int i = 0; i < o.length; i++)
13        Console.scriviStringa(o[i].nome());
14    }
15    public static Oggetto cerca(Oggetto[] o,
16                                String nome) {
17      for (int i = 0; i < o.length; i++) {
18        if (nome.equals(o[i].nome())) {
19          return o[i];
20        }
21      }
22      return null;
```

```
23    }
24    public static void main(String[] args) {
25      Stanza s1 = new Stanza("stanza 1");
26      Stanza s2 = new Stanza("stanza 2");
27      Stanza s3 = new Stanza("stanza 3");
28      Porta p1 = new Porta(s1, s2);
29      s1.collega(Stanza.Direzioni.EST, p1);
30      s2.collega(Stanza.Direzioni.OVEST, p1);
31      Porta p2 = new Porta(s2, s3);
32      s2.collega(Stanza.Direzioni.SUD, p2);
33      s3.collega(Stanza.Direzioni.NORD, p2);
34      Oggetto c1 = new MobileImpl("oggetto1", s1);
35      Oggetto c2 = new MobileImpl("oggetto2", s2);
36      Giocatore giocatore = new Giocatore("Giocatore", s1);
37      boolean ancora = true;
38      while (ancora) {
39        Stanza stanza = (Stanza) giocatore.luogo();
40        Console.scriviStringa("***************************");
41        Console.scriviStringa(stanza.descrivi());
42        Stanza.Direzioni[] dir = stanza.direzioni();
43        if (dir.length > 0) {
44          Console.scriviStr("Puoi andare a: ");
45          for (int i = 0; i < dir.length; i++)
46            Console.scriviStr(dir[i].toString() + " ");
47          Console.nuovaLinea();
48        }
49        Oggetto[] oggetti = stanza.elenco();
50        elenca("Vedi:", oggetti);
51        Console.scriviStringa("Cosa vuoi fare?");
52        String in = Console.leggiStringa();
53        Risultato r = null;
54        if (in.equals("fine")) {
55          ancora = false;
56        } else if (in.equals("vai")) {
57          in = Console.leggiStringa();
58          if (in.equals("nord") || in.equals("sud")
59              || in.equals("est") ||
60              in.equals("ovest")) {
61            Stanza.Direzioni d = Stanza.Direzioni.valueOf(
62                                  in.toUpperCase());
63            r = stanza.vai(giocatore, d);
```

```
64            } else {
65              r = new Risultato(false,
66                                "Non conosco la direzione " + in);
67            }
68          } else if (in.equals("prendi")) {
69            in = Console.leggiStringa();
70            Oggetto o = cerca(oggetti, in);
71            if (o == null) {
72              r = new Risultato(false, "Non vedo " + in + " qui.");
73            } else if (o == giocatore) {
74              r = new Risultato(false,
75                                "Sei, per caso, il Barone di Munchausen?");
76            } else if (!(o instanceof Mobile)) {
77              r = new Risultato(false, "Non si muove.");
78            } else {
79              r = ((Mobile) o).sposta(stanza, giocatore);
80            }
81          } else if (in.equals("lascia")) {
82            in = Console.leggiStringa();
83            Oggetto o = (Oggetto) cerca(giocatore.elenco(), in);
84            if (o == null) {
85              r = new Risultato(false, "Non possiedi " + in);
86            } else {
87              r = ((Mobile) o).sposta(giocatore, stanza);
88            }
89          } else if (in.equals("inventario")) {
90            elenca("Possiedi: ", giocatore.elenco());
91          } else {
92            r = new Risultato(false, "Non capisco.");
93          }
94          if (r != null) {
95            Console.scriviStringa(r.getMessage());
96          }
97        }
98      }
99    }
```

9.9 Porte e chiavi

Vogliamo estendere il gioco presentato nell'esercizio 9.8 nel seguente modo. Alcune porte possono essere inizialmente chiuse e non possono essere attraversate se

non possedendo l'opportuna chiave. Le chiavi sono oggetti che implementano l'interfaccia *Mobile* (per esempio, appartenenti alla classe *MobileImpl* nella soluzione dell'esercizio 9.8).

Definire la classe *PortaChiusa*. La classe *PortaChiusa* deve estendere la classe *Porta*, ridefinendo il metodo *attraversa* in modo che il giocatore possa attraversare la porta solo se possiede la corrispondente chiave. La corrispondenza tra una porta e una chiave viene decisa al momento della creazione di un oggetto di tipo *PortaChiusa*, passando al costruttore un riferimento all'oggetto che rappresenta la chiave.

Soluzione

PortaChiusa.java:

```
1  package cap9.portechiavi;
2
3  import cap9.prenderelasciare.*;
4  import cap6.labirinto.Risultato;
5  public class PortaChiusa extends Porta {
6    boolean aperta = false;
7    Oggetto chiave;
8    public PortaChiusa(
9      Stanza stanza1, Stanza stanza2, Oggetto chiave) {
10     super(stanza1, stanza2);
11     this.chiave = chiave;
12   }
13   public Risultato attraversa(Giocatore giocatore) {
14     Risultato r;
15     if (chiave.luogo() == giocatore) {
16       r = super.attraversa(giocatore);
17       if (r.isOk())
18         r = new Risultato(
19           true, "Apri la porta usando " + chiave.nome());
20     } else
21       r = new Risultato(false, "La porta e' chiusa");
22     return r;
23   }
24 }
```

9.10 Buio e luce

Vogliamo estendere il gioco costruito nell'esercizio 9.8 nel seguente modo. Alcune stanze possono essere *buie*. Quando il giocatore si trova in una stanza buia, non può leggerne la descrizione, né vedere gli oggetti in essa contenuti. Alcuni oggetti possono essere *lampade*. Una lampada può essere accesa o spenta. Una lampada accesa che si trova in una stanza buia (sia se contenuta nella stanza, sia se trasportata da un giocatore che si trova nella stanza) illumina la stanza (quindi, permette di leggerne la descrizione e di vedere gli oggetti contenuti).

Scrivere la classe *Lampada*, con i metodi *void accendi()*, *spegni()* e *boolean accesa()*, di ovvio significato. La classe deve implementare l'interfaccia *Mobile*.

Scrivere la classe *StanzaBuia*, che estende la classe *Stanza*, ridefinendo opportunamente i metodi *descrivi()* e *elenco()*. Tali metodi, ridefiniti, devono verificare la presenza di un oggetto di tipo *Lampada* nella stanza (eventualmente, trasportato dal giocatore), che sia anche acceso. Se tale oggetto viene trovato, i metodi si comportano come quelli della classe *Stanza*. Altrimenti, *descrivi()* restituisce la stringa "Sei al buio", mentre *elenco()* restituisce un array di zero oggetti.

Soluzione

Lampada.java:

```
1  package cap9.buioluce;
2
3  import cap9.prenderelasciare.*;
4  class Lampada extends MobileImpl {
5    protected boolean accesa = false;
6    public Lampada(String nome, Contenitore luogo) {
7      super(nome, luogo);
8    }
9    public boolean accesa() {
10     return accesa;
11   }
12   public void accendi() {
13     accesa = true;
14   }
15   public void spegni() {
16     accesa = false;
17   }
18 }
```

StanzaBuia.java:

```
1  package cap9.buioluce;
2
3  import cap9.prenderelasciare.*;
4  public class StanzaBuia extends Stanza {
5    public StanzaBuia(String descrizione) {
6      super(descrizione);
7    }
8    protected boolean cercaLampada(Oggetto[] o) {
9      boolean ris = false;
10     for (int i = 0; i < o.length; i++)
11       if (
12         o[i] instanceof Lampada &&
13         ((Lampada) o[i]).accesa())
14         return true;
15       else if (
16         o[i] instanceof Contenitore &&
17         cercaLampada(((Contenitore) o[i]).elenco()))
18         return true;
19     return false;
20   }
21   public String descrivi() {
22     if (cercaLampada(oggetti))
23       return super.descrivi();
24     else
25       return "Sei al buio.";
26   }
27   public Oggetto[] elenco() {
28     if (cercaLampada(oggetti))
29       return super.elenco();
30     else
31       return new Oggetto[0];
32   }
33 }
```

9.11 Messaggio cifrato

Definire una classe *Messaggio* in grado di contenere un messaggio testuale (una stringa) e provvista di due metodi astratti *void cifra(String k)* e *void decifra(String k)*, dove *k* è la chiave usata per cifrare e decifrare il messaggio. Definire quindi due sottoclassi che realizzano i seguenti meccanismi di cifratura/decifratura:

ROT n La chiave *k* è una stringa che rappresenta un numero *n*. L'algoritmo cifra il testo sostituendo ogni lettera con quella che la segue di *n* posizioni (circolarmente) nell'insieme delle codifiche unicode. La decifratura avviene in modo simile: ogni lettera del messaggio cifrato viene sostituita con la lettera che la precede di *n* posizioni (circolarmente) nell'insieme delle codifiche unicode.

XOR La chiave *k* è una stringa di lunghezza arbitraria. La cifratura avviene eseguendo lo xor tra i caratteri del testo e i caratteri della chiave. Nel caso in cui il testo abbia una lunghezza superiore alla chiave, i caratteri di quest'ultima vengono utilizzati in modo circolare. La decifratura avviene eseguendo esattamente la stessa procedura sul testo cifrato.

Scrivere infine un programma principale di prova in cui a riferimenti di tipo *Messaggio* vengono assegnati oggetti dei due sottotipi. Provare quindi a stampare a video il testo del messagio dopo averlo cifrato e decifrato.

Soluzione

Messaggio.java:

```
1  package cap9.messaggio;
2
3  public abstract class Messaggio {
4    protected String data;
5    public Messaggio(String s) {
6      data = s;
7    }
8    public abstract void cifra(String chiave);
9    public abstract void decifra(String chiave);
10   public String toString() {
11     return data;
12   }
13 }
```

RotMessaggio.java:

```
1  package cap9.messaggio;
2
3  public class RotMessaggio extends Messaggio {
4    public RotMessaggio(String s) {
5      super(s);
6    }
7
```

```
8     public void cifra(String c) {
9       int n = Integer.parseInt(c);
10      char[] aa = data.toCharArray();
11      for (int i = 0; i < aa.length; i++)
12        aa[i] = (char) ((aa[i] + n) % 65536);
13      data = new String(aa);
14    }
15
16    public void decifra(String c) {
17      int n = Integer.parseInt(c);
18      n = 65536 - n;
19      cifra(Integer.toString(n));
20    }
21  }
```

XorMessaggio.java:

```
1   package cap9.messaggio;
2
3   public class XorMessaggio extends Messaggio {
4     public XorMessaggio(String s) {
5       super(s);
6     }
7     public void cifra(String c) {
8       char[] aa = data.toCharArray();
9       char[] cc = c.toCharArray();
10      int x = 0;
11      for (int i = 0; i < aa.length; i++)
12        aa[i] = (char) (aa[i] ^ cc[x = (x + 1) % cc.length]);
13      data = new String(aa);
14    }
15    public void decifra(String c) {
16      cifra(c);
17    }
18  }
```

Prova.java:

```
1   package cap9.messaggio;
2
3   import IngressoUscita.Console;
4   public class Prova {
```

```
5     public static void main(String[] args) {
6       Messaggio m1 = new RotMessaggio("Testo segreto...");
7       Console.scriviStringa(m1.toString());
8       m1.cifra("32");
9       Console.scriviStringa(m1.toString());
10      m1.decifra("32");
11      Console.scriviStringa(m1.toString());
12      Messaggio m2 =
13        new XorMessaggio("Messaggio importante...");
14      Console.scriviStringa(m2.toString());
15      m2.cifra("abracadabra");
16      Console.scriviStringa(m2.toString());
17      m2.decifra("abracadabra");
18      Console.scriviStringa(m2.toString());
19    }
20  }
```

10. Eccezioni

10.1 Ricerca in un vettore

Scrivere una funzione che cerca un valore intero in un vettore di interi, ordinato in senso crescente. Sia il valore che il vettore sono argomenti della funzione. La funzione restituisce la posizione del valore nel vettore, ma lancia una eccezione se rileva che il vettore passatole come argomento non è ordinato o se il valore non è presente nel vettore. Scrivere quindi un programma principale che legge da tastiera un vettore e un valore e chiama la funzione di ricerca, intercettando eventuali eccezioni.

Soluzione

Ricerca.java:

```
1  package cap10.ricerca;
2
3  import IngressoUscita.Console;
4  public class Ricerca {
5    static int ricerca(int val,
6                       int[] vec) throws Exception {
7      boolean trovato = false;
8      int i = 0;
9      while (!trovato && (i < vec.length)
10            && (vec[i] <= val)) {
11       if ((i > 0) && !(vec[i] >= vec[i - 1])) {
12         throw new Exception("Vettore non ordinato");
13       }
14       if (vec[i] == val) {
```

```
15          trovato = true;
16        } else {
17          i++;
18        }
19      }
20      if (!trovato) {
21        throw new Exception("Valore non trovato");
22      }
23      return i;
24    }
25    public static void main(String[] args) {
26      Console.scriviStringa("Quanti elementi? ");
27      int n = Console.leggiIntero();
28      int[] vettore = new int[n];
29      for (int i = 0; i < n; i++) {
30        Console.scriviStringa("Elemento n. " + i + "? ");
31        vettore[i] = Console.leggiIntero();
32      }
33      Console.scriviStringa("Valore da cercare? ");
34      int valore = Console.leggiIntero();
35      try {
36        int posizione = ricerca(valore, vettore);
37        Console.scriviStringa("Valore trovato in posizione: "
38                              +
39                              posizione);
40      } catch (Exception e) {
41        Console.scriviStringa("Errore: " + e.getMessage());
42      }
43      Console.scriviStringa("Fine");
44    }
45  }
```

10.2 Pila con eccezioni

Modificare la classe Pila dell'esercizio 9.1 nel seguente modo:

- le situazioni anomale (estrazione da pila vuota, inserimento in pila piena, inserimento del valore *null*) devono essere segnalate da eccezioni.
- aggiungere un metodo *accresci()*, senza parametri, che raddoppia le dimensioni della pila.

- in *MenuPila*, gestire la situazione anomala di inserimento in pila piena raddoppiando la dimensione della pila.

Soluzione

PilaVuotaException.java:

```
1  package cap10.pila;
2
3  public class PilaVuotaException extends Exception {
4  }
```

PilaPienaException.java:

```
1  package cap10.pila;
2
3  public class PilaPienaException extends Exception {
4  }
```

ParametroErratoException.java:

```
1  package cap10.pila;
2
3  public class ParametroErratoException extends
4    Exception {}
```

Pila.java:

```
1  package cap10.pila;
2
3  public class Pila {
4    static private final int DEFAULT_SIZE = 10;
5    protected int size;
6    protected int top;
7    protected Object[] v;
8    public Pila() {
9      this(DEFAULT_SIZE);
10   }
11   public Pila(int s) {
12     size = s;
13     top = -1;
14     v = new Object[size];
15   }
```

```
16    public boolean isEmpty() {
17      return top == -1;
18    }
19    public boolean isFull() {
20      return top == (size - 1);
21    }
22    public void push(Object elem)
23    throws PilaPienaException, ParametroErratoException {
24      if (isFull())
25        throw new PilaPienaException();
26      if (elem == null)
27        throw new ParametroErratoException();
28      top++;
29      v[top] = elem;
30    }
31    public Object pop() throws PilaVuotaException {
32      if (isEmpty())
33        throw new PilaVuotaException();
34      Object result = v[top];
35      top--;
36      return result;
37    }
38    public Object top() throws PilaVuotaException {
39      if (isEmpty())
40        throw new PilaVuotaException();
41      return v[top];
42    }
43    public void accresci() {
44      int newSize = size * 2;
45      Object[] tmp = new Object[newSize];
46      System.arraycopy(v, 0, tmp, 0, size);
47      v = tmp;
48      size = newSize;
49    }
50    public String toString() {
51      String ls = System.getProperty("line.separator");
52      String s = "---" + ls;
53      if (!isEmpty()) {
54        for (int i = top; i >= 0; i--)
55          s += (v[i] + ls);
56      }
```

```
57      s += "---";
58      return s;
59    }
60    public boolean equals(Object o) {
61      if ((o == null) || !(o instanceof Pila))
62        return false;
63      Pila p = (Pila) o;
64      if ((p.top != top) || (p.size != size))
65        return false;
66      for (int i = top; i >= 0; i--)
67        if (!v[i].equals(p.v[i]))
68          return false;
69      return true;
70    }
71  }
```

MenuPila.java:

```
1   package cap10.pila;
2
3   import IngressoUscita.Console;
4   class MenuPila {
5     private static void stampaMenu() {
6       Console.scriviStringa("Scegli:");
7       Console.scriviStringa("1 Inserisci una stringa nella pila");
8       Console.scriviStringa("2 Estrai una stringa dalla pila");
9       Console.scriviStringa("3 Stampa il contenuto della pila");
10      Console.scriviStringa("4 Verifica se la pila e' vuota");
11      Console.scriviStringa("5 Verifica se la pila e' piena");
12      Console.scriviStringa("6 Esci");
13    }
14    public static void main(String[] args) {
15      Pila pila = new Pila();
16      for (;;) {
17        stampaMenu();
18        int s = Console.leggiIntero();
19        try {
20          switch (s) {
21          case 1:
22            Console.scriviStringa("Stringa da inserire ?");
23            String x = Console.leggiStringa();
24            try {
```

```
25              pila.push(x);
26            } catch (PilaPienaException e) {
27              Console.scriviStringa("pila piena: aumento le dimensioni");
28              pila.accresci();
29              pila.push(x);
30            }
31            break;
32          case 2:
33            Object o = pila.pop();
34            Console.scriviStringa("La stringa estratta e' " + o);
35            break;
36          case 3:
37            Console.scriviStringa(pila.toString());
38            break;
39          case 4:
40            Console.scriviStringa("Pila vuota: " +
41                                  (pila.isEmpty() ? "vero" : "falso"));
42            break;
43          case 5:
44            Console.scriviStringa("Pila piena: " +
45                                  (pila.isFull() ? "vero" : "falso"));
46            break;
47          case 6:
48            System.exit(0);
49          default:
50            Console.scriviStringa("Scelta non valida");
51          }
52        } catch (PilaVuotaException e) {
53          Console.scriviStringa("Pila vuota!");
54        } catch (Exception e) {
55          Console.scriviStringa("Errore:" + e.getMessage());
56        }
57      }
58    }
59  }
```

Note

Nella classe *MenuPila*, l'eccezione *PilaPienaException* viene intercettata nel blocco *try-catch* più interno, alle linee 28–36, in modo da eseguire *accresci()* e ripetere l'operazione di *push()* (che, a questo punto, andrà sicuramente a buon fine).

Tutte le altre eccezioni vengono intercettate dal blocco *try-catch* più esterno (23–67), in quanto l'esercizio non richiede di eseguire azioni specifiche in quei casi. In questo caso, si è scelto, semplicemente, di stampare un messaggio di errore e far proseguire il programma. Si noti, comunque, che, per poter stampare un messaggio di errore specifico per l'eccezione *PilaVuotaException*, è necessario intercettare tale eccezione (linee 62–64) prima dell'eccezione generica *Exception* (linee 65–66), in quanto *PilaVuotaException* non porta con sé alcun messaggio testuale (file *PilaVuotaException.java*).

10.3 Programma errato

Realizzare le classi *EccezioneA*, *EccezioneB*, *EccezioneC* secondo la gerarchia indicata in Fig. 10.1

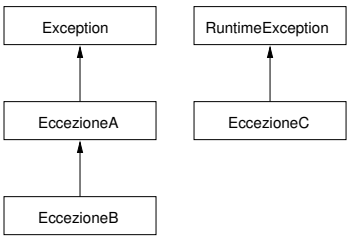


Figura 10.1: Gerachia di eccezioni

Scrivere una classe *ClasseErr* che simula il comportamento di una classe contenente degli errori di programmazione. La classe deve essere dotata dei metodi seguenti.

- *void metodo1()*: nel 10% dei casi lancia una eccezione di tipo *EccezioneC* e nel restante 90% dei casi non fa niente.

- *void metodo2()*: nel 50% dei casi lancia una eccezione di tipo *EccezioneA* e nel restante 50% lancia una eccezione di tipo *EccezioneB*.

Scrivere infine un programma di prova che crea una istanza di *ClasseErr*, visualizza un menu attraverso il quale l'utente può specificare quale dei due metodi vuole eseguire, cattura tutte le eccezioni che si possono verificare e stampa un messaggio a video distinto per ognuna di esse (senza che il programma venga interrotto).

Suggerimenti

Per eseguire una operazione solo nel $x\%$ dei casi, si può generare un numero casuale c compreso tra 0 e 1 (tramite la funzione *Math.random()*) e controllare che $c \leq x$.

Soluzione

EccezioneA.java:

```
1  package cap10.progerr;
2
3  public class EccezioneA extends Exception {
4    public EccezioneA() {
5      super();
6    }
7
8    public EccezioneA(String s) {
9      super(s);
10   }
11 }
```

EccezioneB.java:

```
1  package cap10.progerr;
2
3  public class EccezioneB extends EccezioneA {
4    public EccezioneB() {
5      super();
6    }
7
8    public EccezioneB(String s) {
9      super(s);
10   }
11 }
```

EccezioneC.java:

```
1  package cap10.progerr;
2
3  public class EccezioneC extends RuntimeException {
4    public EccezioneC() {
5      super();
```

```
6     }
7
8     public EccezioneC(String s) {
9       super(s);
10    }
11  }
```

ClasseErr.java:

```
1   package cap10.progerr;
2
3   public class ClasseErr {
4     public void metodo1() {
5       double r = Math.random();
6       if (r <= 0.1)
7         throw new EccezioneC("Problema C in metodo 1...");
8     }
9     public void metodo2() throws EccezioneA {
10      double r = Math.random();
11      if (r <= 0.5)
12        throw new EccezioneA("Problema A in metodo 2...");
13      else
14        throw new EccezioneB("Problema B in metodo 2...");
15    }
16  }
```

Prova.java:

```
1   package cap10.progerr;
2
3   import IngressoUscita.Console;
4   public class Prova {
5     public static void main(String[] args) {
6       ClasseErr ce = new ClasseErr();
7       int i;
8       do {
9         Console.scriviStringa("1) metodo1");
10        Console.scriviStringa("2) metodo2");
11        Console.scriviStringa("3) fine");
12        i = Console.leggiIntero();
13        try {
14          switch (i) {
```

```
15            case 1:
16              ce.metodo1();
17              break;
18            case 2:
19              ce.metodo2();
20              break;
21            }
22          } catch (EccezioneB eb) {
23            Console.scriviStringa("Catturata eccezione B:");
24            Console.scriviStringa(eb.getMessage());
25          } catch (EccezioneA ea) {
26            Console.scriviStringa("Catturata eccezione A:");
27            Console.scriviStringa(ea.getMessage());
28          } catch (EccezioneC ec) {
29            Console.scriviStringa("Catturata eccezione C:");
30            Console.scriviStringa(ec.getMessage());
31          }
32        } while (i != 3);
33      }
34    }
```

Note

Il fatto che, nella classe *Prova*, il blocco *try-catch* sia contenuto all'interno del ciclo fa sì che il programma non termini quando viene catturata una eccezione. *EccezioneC* è sottoclasse di *RuntimeException* pertanto il programmatore non è obbligato a gestirla (le righe 32–35 nel file *Prova.java* possono anche essere eliminate, ma in tale caso il programma terminerebbe al verificarsi della situazione anomala associata a *EccezioneC*). Il *catch* relativo ad *EccezioneB*, se presente, deve precedere quello relativo a *EccezioneA* in quanto quest'ultima è sua superclasse. Il blocco *catch* relativo ad *EccezioneB* potrebbe essere omesso (è sufficiente il *catch* relativo a *EccezioneA*), ma in tal caso il programma non saprebbe distinguere tra i due tipi di eccezione.

10.4 Controllo sintassi

Scrivere una classe *Sintassi* che è in grado di controllare se una data stringa di ingresso appartiene al linguaggio generato dal simbolo *Exp* nella seguente grammatica:

Var ::= “A” | “B” | ··· | “Z”
Op ::= “+” | “-” | “*” | “/”
Exp ::= *Var* | “(” *Exp Op Exp* “)”

Per esempio, appartengono al linguaggio (rispettano la sintassi) le stringhe “B” e “((A+Z)*(B-W))’’, ma non le stringhe “(A+” o “A+(B*C)” (quest’ultima non appartiene al linguaggio perché, se la stringa non comincia con “(”, deve essere composta solo da *Var*).

Scrivere quindi un programma principale che legge ciclicamente una linea da tastiera e scrive “Sintassi OK” se la sintassi è rispettata, e “Errore di sintassi” altrimenti. Il programma termina quando la linea letta coincide con la stringa “fine”.

Suggerimenti

Un possibile modo di organizzare la classe *Sintassi* è quello di prevedere una funzione *skipX* per ogni simbolo non terminale *X* della grammatica. La funzione *skipX* deve ricevere una stringa *s* in ingresso, controllare che *s* contenga, all’inizio, una sequenza *l* che rispetta le regole di *X*, quindi restituire la sottostringa s' di *s*, una volta che sia stata tolta la sequenza *l* così trovata. Nel controllare la sintassi, ogni funzione può chiamare, anche ricorsivamente, le altre. Ogni funzione lancerà una eccezione se rileva che la sintassi non è rispettata. La sintassi è corretta se, alla fine, tutta la stringa iniziale viene “consumata” e non è stata lanciata nessuna eccezione.

Soluzione

SyntaxError.java:

```
1  package cap10.contrsint;
2
3  public class SyntaxError extends Exception {
4    public SyntaxError(String s) {
5      super(s);
6    }
7  }
```

Sintassi.java:

```
1  package cap10.contrsint;
2
3  public class Sintassi {
4    public static String skipVar(String s) throws
5      SyntaxError {
6      char c = s.charAt(0);
```

```
7       if (!((c >= 'A') && (c <= 'Z'))) {
8         throw new SyntaxError("'" + c +
9                               "' non e' una variabile");
10      }
11      return s.substring(1);
12    }
13
14    public static String skipOp(String s) throws
15      SyntaxError {
16      char c = s.charAt(0);
17      if ((c != '+') && (c != '-') && (c != '*')
18          && (c != '/')) {
19        throw new SyntaxError("'" + c +
20                              "' non e' un operatore");
21      }
22      return s.substring(1);
23    }
24
25    public static String skipExpr(String s) throws
26      SyntaxError {
27      if (s.charAt(0) != '(') {
28        return skipVar(s);
29      }
30      s = skipExpr(skipOp(skipExpr(s.substring(1))));
31      if (s.charAt(0) != ')') {
32        throw new SyntaxError("manca parentesi chiusa");
33      }
34      return s.substring(1);
35    }
36  }
```

Parser.java:

```
1   package cap10.contrsint;
2
3   import IngressoUscita.Console;
4   public class Parser {
5     private static void check(String s) throws
6       SyntaxError {
7       s = Sintassi.skipExpr(s);
8       if (s.length() > 0) {
9         throw new SyntaxError("'" + s + "' non riconosciuto");
```

```
10        }
11      }
12      public static void main(String[] args) {
13        String line;
14        for (;;) {
15          line = Console.leggiLinea();
16          if (line.equals("fine")) {
17            break;
18          }
19          try {
20            check(line);
21            Console.scriviStringa("Sintassi OK");
22          } catch (SyntaxError e) {
23            Console.scriviStringa("Errore di sintassi: " +
24                                  e.getMessage());
25          } catch (Exception e) {
26            Console.scriviStringa(e.getMessage());
27          }
28        }
29      }
30    }
```

Note

Notare che, mano a mano che le funzioni si addentrano nella stringa da analizzare, possono succedere due cose che rivelano che la stringa iniziale era scorretta:

1. si incontra un carattere che non appartiene a quelli possibili in quel momento. Per esempio, dopo aver visto una sottoespressione, si cerca un operatore, che può essere solo "`+`", "`-`", "`*`" o "`/`". Un qualunque carattere diverso da questi è sicuramente un errore;

2. la stringa finisce prematuramente. Per esempio, la stringa è "`(A+`".

Gli errori del tipo 1 vengono segnalati da eccezioni di tipo *SyntaxError* nella soluzione proposta. La soluzione proposta, invece, non controlla esplicitamente la presenza di errori del tipo 2. Tali errori, però, vengono comunque correttamente rilevati, sfruttando il Java run-time environment: se, ad un certo punto, la sintassi prevede che la stringa debba contenere un carattere e invece la stringa è finita, verrà sicuramente invocata la funzione *s.charAt(0)* su una stringa *s* vuota. A quel punto, il Java run-time environment solleverà una eccezione di *IndexOutOfBoundsException*.

10.5 Controllo sintassi (su più linee)

Modificare la classe *Parser* della soluzione dell'esercizio 10.4 nel seguente modo. Quando viene rilevato un errore di fine prematura della linea (per esempio, quando viene inserita la linea "`(A+`"), deve essere data la possibilità di inserire una nuova linea che si aggiunge alla precedente (in pratica, si deve dare la possibilià all'utente di scrivere una espressione completa su più linee, e non su un'unica linea).

Soluzione

Parser.java:

```
1  package cap10.contrsint2;
2
3  import IngressoUscita.Console;
4  import cap10.contrsint.*;
5  public class Parser {
6    private static void check(String s) throws
7      SyntaxError {
8      s = Sintassi.skipExpr(s);
9      if (s.length() > 0) {
10       throw new SyntaxError("'" + s + "' non riconosciuto");
11     }
12   }
13   public static void main(String[] args) {
14     String line;
15     String oldline = "";
16     for (;;) {
17       line = Console.leggiLinea();
18       if (line.equals("fine")) {
19         break;
20       }
21       line = oldline + line;
22       try {
23         Console.scriviStringa("Controllo la linea '" + line +
24                               "'");
25         check(line);
26         Console.scriviStringa("Sintassi OK");
27         oldline = "";
28       } catch (SyntaxError e) {
29         Console.scriviStringa("Errore di sintassi: " +
30                               e.getMessage());
```

```
31            oldline = "";
32          } catch (IndexOutOfBoundsException e) {
33            Console.scriviStringa("Completare la linea");
34            oldline = line;
35          } catch (Exception e) {
36            Console.scriviStringa(e.getMessage());
37          }
38        }
39      }
40    }
```

Note

La possibilità di scrivere la stringa su più linee è realizzata nelle linee 19 e 32–34. Quando viene generata l'eccezione *IndexOutOfBoundException*, vuol dire che la linea precedente era terminata prematuramente (cfr. le Note dell'esercizio 10.4). In quel caso, la variabile *oldline* assume il valore della stringa corrente (linea 34) e, al prossimo ciclo, la nuova stringa letta da tastiera viene concatenata alla precedente (linea 19).

10.6 Controllo sintassi (debug)

Modificare la soluzione dell'esercizio 10.4 nel seguente modo. Per ogni stringa introdotta in ingresso deve essere prodotta una stampa di debug che, nel caso di sintassi corretta, deve essere simile al seguente esempio (in cui la stringa di ingresso è "`(A+(B/C)`"):

```
   skipExpr("(A+(B/C))")
      skipExpr("A+(B/C))")
         skipVar("A+(B/C))")
      skipOp("+(B/C))")
      skipExpr("(B/C))")
         skipExpr("B/C))")
            skipVar("B/C))")
         skipOp("/C))")
         skipExpr("C))")
            skipVar("C))")
Sintassi OK
```

Nel caso di sintassi errata, l'uscita di debug deve essere simile al seguente esempio (in cui la stringa di ingresso è "`(A+*)`"):

```
   skipExpr("(A+*)")
      skipExpr("A+*)")
         skipVar("A+*)")
      skipOp("+*)")
      skipExpr("*)")
         skipVar("*)")
         Errore!
      Errore!
   Errore!
Errore di sintassi: '*' non e' una variabile
```

Suggerimenti

Per produrre l'indentazione corretta nell'uscita, si può usare una variabile *nesting* che conta il livello di annidamento delle funzioni: la si incrementa ogni volta che si entra in una delle funzioni del parser, e la si decrementa ogni volta che si esce. Fare attenzione ai vari modi in cui si può uscire da una funzione....

Soluzione

Sintassi.java:

```
1  package cap10.contrsint3;
2
3  import IngressoUscita.Console;
4  import cap10.contrsint.SyntaxError;
5  public class Sintassi {
6    private static int nesting = 0;
7    private static void stampa(String s) {
8      for (int i = 0; i < nesting; i++)
9        Console.scriviStr("  ");
10     Console.scriviStringa(s);
11   }
12   public static String skipVar(String s) throws
13     Exception {
14     nesting++;
15     stampa("skipVar(\"" + s + "\")");
16     try {
17       char c = s.charAt(0);
18       if (!((c >= 'A') && (c <= 'Z'))) {
19         throw new SyntaxError("'" + c +
20                               "' non e' una variabile");
```

```
21          }
22        } catch (Exception e) {
23          stampa("Errore!");
24          throw e;
25        } finally {
26          nesting--;
27        }
28        return s.substring(1);
29      }
30      public static String skipOp(String s) throws
31        Exception {
32        nesting++;
33        stampa("skipOp(\"" + s + "\")");
34        try {
35          char c = s.charAt(0);
36          if ((c != '+') && (c != '-') && (c != '*')
37              && (c != '/')) {
38            throw new SyntaxError("'" + c +
39                                  "' non e' un operatore");
40          }
41        } catch (Exception e) {
42          stampa("Errore!");
43          throw e;
44        } finally {
45          nesting--;
46        }
47        return s.substring(1);
48      }
49      public static String skipExpr(String s) throws
50        Exception {
51        nesting++;
52        stampa("skipExpr(\"" + s + "\")");
53        try {
54          if (s.charAt(0) != '(') {
55            return skipVar(s);
56          }
57          s = skipExpr(skipOp(skipExpr(s.substring(1))));
58          if (s.charAt(0) != ')') {
59            throw new SyntaxError("manca parentesi chiusa");
60          }
61        } catch (Exception e) {
```

```
62          stampa("Errore!");
63          throw e;
64        } finally {
65          nesting--;
66        }
67        return s.substring(1);
68      }
69    }
```

Note

Per decrementare la variabile *nesting* ogni volta che si esce da una funzione (sia nel caso normale, sia nel caso in cui viene sollevata una eccezione), si è fatto uso dei blocchi *finally* alle linee 26–28, 46–48 e 65–67.

11. Ingresso e uscita

11.1 Copia di file

Si scriva un programma per la copia di file. Il nome del file sorgente e del file destinazione sono specificati da riga di comando, per esempio *java Copia fileSorgente fileDestinazione*. Nel caso in cui il file destinazione sia già esistente il programma deve chiedere conferma all'utente prima di sovrascriverne il contenuto.

Soluzione

Copia.java:

```
1  package cap11.copiafile;
2
3  import IngressoUscita.*;
4  import java.io.*;
5  public class Copia {
6    private File sorg;
7    private File dest;
8    public Copia(String s, String d) {
9      sorg = new File(s);
10     dest = new File(d);
11   }
12   public void copia() throws IOException {
13     if (dest.exists()) {
14       Console.scriviStringa("Vuoi sovrascrivere il file " +
15                             dest.getName() +
16                             " (s/n) ?");
17       String s = Console.leggiStringa();
```

```
18        if (!s.startsWith("s")) {
19          return;
20        }
21      }
22      byte[] buffer = new byte[1024];
23      BufferedInputStream bis = null;
24      BufferedOutputStream bos = null;
25      try {
26        bis = new BufferedInputStream(new FileInputStream(
27                                        sorg));
28        bos = new BufferedOutputStream(new FileOutputStream(
29                                         dest));
30        int letti;
31        while ((letti = bis.read(buffer)) > 0)
32          bos.write(buffer, 0, letti);
33      } finally {
34        try {
35          if (bos != null) {
36            bos.close();
37          }
38          if (bis != null) {
39            bis.close();
40          }
41        } catch (IOException ioe) {
42        }
43      }
44    }
45    public static void main(String[] args) {
46      if (args.length != 2) {
47        Console.scriviStringa("Uso: java Copia fileSorg fileDest");
48        return;
49      }
50      Copia cp = new Copia(args[0], args[1]);
51      try {
52        cp.copia();
53      } catch (FileNotFoundException fnfe) {
54        Console.scriviStringa("Problema durante apertura file: "
55                              +
56                              fnfe.getMessage());
57      } catch (IOException ioe) {
58        Console.scriviStringa("Errore durante la copia: " +
```

```
59                                  ioe.getMessage());
60        }
61      }
62    }
```

Note

Il costruttore di *FileInputStream* (righe 26–27) genera una eccezione di tipo *FileNotFoundException* (sottotipo di *IOException*) nel caso in cui il file specificato non esista, sia una directory, o non sia permesso leggerne il contenuto. Il metodo *copia()* propaga l'eventuale eccezione al chiamante che stampa un messaggio.

11.2 Cerca

Realizzare un programma che cerca all'interno di un file di testo tutte le righe che contengono una determinata stringa e le stampa a video. La stringa da cercare e il nome del file vengono passati da riga di comando, per esempio *java Cerca pippo prova.txt*.

Soluzione

Cerca.java:

```
1   package cap11.cerca;
2
3   import IngressoUscita.*;
4   import java.io.*;
5   public class Cerca {
6     private String str;
7     private File file;
8     public Cerca(String s, String f) {
9       str = s;
10      file = new File(f);
11    }
12    public void avvio() {
13      if (file.isDirectory()) {
14        Console.scriviStringa("Il file scelto e' una directory");
15        return;
16      }
17      String s;
18      BufferedReader br = null;
```

```
19      try {
20        br = new BufferedReader(new FileReader(file));
21        while ((s = br.readLine()) != null) {
22          if (s.indexOf(str) != -1) {
23            Console.scriviStringa(s);
24          }
25        }
26      } catch (FileNotFoundException fnfe) {
27        Console.scriviStringa("Il file non esiste");
28      } catch (IOException ioe) {
29        Console.scriviStringa("Problemi durante l'accesso al file");
30      } finally {
31        try {
32          if (br != null) {
33            br.close();
34          }
35        } catch (IOException e) {
36        }
37      }
38    }
39    public static void main(String[] args) {
40      if (args.length != 2) {
41        Console.scriviStringa("Uso: java Cerca stringa nomeFile");
42        return;
43      }
44      Cerca c = new Cerca(args[0], args[1]);
45      c.avvio();
46    }
47  }
```

11.3 Stampa file

Realizzare un programma che:

1. prende in ingresso, da riga di comando, un path;
2. stampa a video la lista dei file contenuti nella directory identificata dal path (se il path non identifica una directory stampare un messaggio di errore);
3. chiede all'utente di scegliere uno dei file della lista;
4. se il file selezionato è un file ordinario, ne stampa a video il contenuto.

Il programma non deve fare uso della classe *Console*.

Soluzione

StampaFile.java:

```
1  package cap11.stampafile;
2
3  import java.io.*;
4  public class StampaFile {
5    private File dir;
6    private File[] listaFile;
7    public StampaFile() {
8      this(".");
9    }
10   public StampaFile(String d) {
11     dir = new File(d);
12     if (dir.isDirectory())
13       listaFile = dir.listFiles();
14   }
15   public void stampaLista() {
16     if (listaFile == null) {
17       System.out.println(
18         "Il path specificato " +
19         "non identifica una directory.");
20       System.exit(1);
21     }
22     System.out.println("Elenco dei file: ");
23     for (int i = 0; i < listaFile.length; i++)
24       System.out.println(
25         (i + 1) + ") " + listaFile[i].getName());
26   }
27   public int sceltaOpzione() {
28     int i = 0;
29     BufferedReader br =
30       new BufferedReader(new InputStreamReader(System.in));
31     for (;;) {
32       System.out.println("Quale file vuoi visualizzare ?");
33       try {
34         String s = br.readLine();
35         i = Integer.parseInt(s);
36       } catch (IOException e) {
```

```
37          e.printStackTrace();
38        } catch (NumberFormatException nfe) {}
39        if ((i < 1) || (i > listaFile.length))
40          System.out.println("Scelta non valida.");
41        else
42          return i;
43      }
44    }
45    public void stampaFile(int sel) {
46      sel--;
47      File f = listaFile[sel];
48      if (f.isDirectory()) {
49        System.out.println(
50          "Il file scelto e' una directory.");
51        return;
52      }
53      String s = null;
54      BufferedReader br = null;
55      try {
56        br = new BufferedReader(new FileReader(f));
57        while ((s = br.readLine()) != null)
58          System.out.println(s);
59      } catch (FileNotFoundException fnfe) {
60        System.out.println("Il file non e' stato trovato.");
61      } catch (IOException ioe) {
62        System.out.println(
63          "Problemi durante la lettura da file.");
64      } finally {
65        try {
66          if (br != null)
67            br.close();
68        } catch (IOException ioe) {}
69      }
70    }
71    public static void main(String[] args) {
72      StampaFile sf;
73      if (args.length == 0)
74        sf = new StampaFile();
75      else
76        sf = new StampaFile(args[0]);
77      sf.stampaLista();
```

```
78        sf.stampaFile(sf.sceltaOpzione());
79      }
80    }
```

11.4 Rimuovi vocali

Si scriva una classe filtro che rimuove le vocali dallo stream di testo a cui è applicata. Quindi, usare tale classe per rimuovere le vocali da un file di testo, stampando a video il risultato.

Suggerimenti

Poiché si tratta di un filtro che opera su stream di caratteri, può essere conveniente estendere la classe *FilterReader*.

Soluzione

RimuoviVocali.java:

```
1   package cap11.rimuovi;
2
3   import java.io.*;
4   public class RimuoviVocali extends FilterReader {
5     public RimuoviVocali(Reader rr) {
6       super(rr);
7     }
8     public int read() throws IOException {
9       int i;
10      boolean continua;
11      do {
12        i = in.read();
13        char c = (char) i;
14        switch (c) {
15        case 'a':
16        case 'A':
17        case 'e':
18        case 'E':
19        case 'i':
20        case 'I':
21        case 'o':
22        case 'O':
```

```
23        case 'u':
24        case 'U':
25          continua = true;
26          break;
27        default:
28          continua = false;
29        }
30      } while ((i != -1) && continua);
31      return i;
32    }
33    public int read(char[] cbuf, int off,
34                    int len) throws IOException {
35      int n = 0;
36      int i;
37      do {
38        i = read();
39        if (i != -1) {
40          cbuf[off + n++] = (char) i;
41        }
42      } while ((i != -1) && (n < len));
43      if ((i == -1) && (n == 0)) {
44        return -1;
45      }
46      return n;
47    }
48  }
```

Test.java:

```
1   package cap11.rimuovi;
2
3   import java.io.*;
4   public class Test {
5     public static void main(String[] args) {
6       if (args.length != 1) {
7         System.out.println("Uso: java Test filename");
8         return;
9       }
10      try {
11        BufferedReader br = new BufferedReader(
12          new RimuoviVocali(
13            new FileReader(args[0])));
```

```
14          String s;
15          while ((s = br.readLine()) != null)
16            System.out.println(s);
17        } catch (IOException e) {
18          e.printStackTrace();
19        }
20      }
21    }
```

11.5 Mappa del labirinto

Con riferimento all'esercizio 6.11, scrivere una classe *Mappa* che permetta di creare un labirinto leggendone la descrizione da file. La classe *Mappa* deve avere i seguenti metodi:

- *Mappa(String fstanze, String fmappa)*: costruttore, a cui vengono passati i nomi di due file: il *file delle stanze* e il *file della mappa*.
- *Stanza creaLabirinto()*: crea un labirinto leggendo la descrizione dai due file, il cui nome è stato specificato tramite il costruttore dell'oggetto. Restituisce un riferimento alla stanza iniziale del labirinto.

Il file delle stanze deve avere il seguente formato: per ogni stanza, ci deve essere una linea contenente esclusivamente un numero (che serve ad identificare la stanza nel file della mappa), subito seguita da una serie di linee, che corrispondono alla descrizione della stanza. La descrizione della stanza deve essere terminata da una linea contenente, esclusivamente, i caratteri "%%". Per esempio, un possibile file delle stanze, può essere:

```
1
descrizione della stanza 1
%%
2
descrizione della stanza 2
continuazione della descrizione
della stanza 2
%%
3
descrizione della stanza 3
%%
```

La stanza iniziale del labirinto è la prima stanza definita nel file delle stanze.

Il file della mappa deve contenere una sequenza (non importa se su più linee) di terne "*sorgente direzione destinazione*", dove *sorgente* e *destinazione* sono numeri naturali (che si riferiscono ad una delle stanze precedentemente definite nel file delle stanze), mentre *direzione* può assumere i valori "nord", "sud", "est" o "ovest". Il significato di una terna è che la stanza *sorgente* deve essere collegata tramite una porta, in direzione *direzione*, alla stanza *destinazione* (e, quindi, la stanza *destinazione* deve essere collegata, tramite la stessa porta, alla stanza *sorgente*, nella direzione opposta a *direzione*). Un esempio di file della mappa può essere:

```
1 nord 2
1 est 3
```

Soluzione

Mappa.java:

```
1  package cap11.mappalabirinto;
2
3  import IngressoUscita.Console;
4  import cap6.labirinto.*;
5  import java.io.*;
6  import java.util.*;
7  class ErroreMappaException extends Exception {
8    public ErroreMappaException(String e) {
9      super(e);
10   }
11 }
12 public class Mappa {
13   public static final int MAX_STANZE = 100;
14   private String fstanze;
15   private String fmappa;
16   private Stanza[] stanze = new Stanza[MAX_STANZE];
17   public Mappa(String s, String m) {
18     fstanze = s;
19     fmappa = m;
20   }
21   public Stanza creaLabirinto() throws
22     ErroreMappaException {
23     String s;
24     BufferedReader br = null;
25     Scanner tas = null;
26     Stanza init = null;
```

```
27      Console.scriviStringa("Leggo il file " + fstanze);
28      try {
29        br = new BufferedReader(new FileReader(fstanze));
30        int linea = 0;
31        for (;;) {
32          s = br.readLine();
33          linea++;
34          int i = Integer.parseInt(s);
35          if ((i < 0) || (i > MAX_STANZE))
36            throw new ErroreMappaException(
37              fstanze + "(" + linea + "): numero " + i +
38              " fuori range");
39          if (stanze[i] != null)
40            throw new ErroreMappaException(
41              fstanze + "(" + linea + "): stanza " + i +
42              " ridefinita");
43          String des = "";
44          for (;;) {
45            String ln = br.readLine();
46            linea++;
47            if (ln.equals("%%"))
48              break;
49            des += (System.getProperty("line.separator") + ln);
50          }
51          if (des.equals(""))
52            throw new ErroreMappaException(
53              fstanze + "(" + linea + "): stanza " + i +
54              " senza descrizione");
55          stanze[i] = new Stanza(des);
56          if (init == null)
57            init = stanze[i];
58        }
59      } catch (Exception e) {}
60      finally {
61        try {
62          br.close();
63        } catch (IOException e) {}
64      }
65      if (init == null)
66        throw new ErroreMappaException(
67          fstanze + ": nessuna stanza definita");
```

```
68        Console.scriviStringa("Leggo il file " + fmappa);
69        try {
70          br = new BufferedReader(new FileReader(fmappa));
71          tas = new Scanner(br);
72          for (;;) {
73            int src = tas.nextInt();
74            if ((src < 0) || (src > MAX_STANZE))
75              throw new ErroreMappaException(
76                fmappa + ": numero " + src + " fuori range");
77            if (stanze[src] == null)
78              throw new ErroreMappaException(
79                fmappa + ": stanza " + src + " non esiste");
80            String sdir = tas.next();
81            Stanza.Direzioni dir1;
82            try {
83              dir1 = Stanza.Direzioni.valueOf(
84                        sdir.toUpperCase());
85            } catch (IllegalArgumentException e) {
86              throw new ErroreMappaException(
87                fmappa + ": direzione " + sdir +
88                " sconosciuta");
89            }
90            Stanza.Direzioni dir2 = null;
91            switch (dir1) {
92            case NORD:
93              dir2 = Stanza.Direzioni.SUD;
94              break;
95            case SUD:
96              dir2 = Stanza.Direzioni.NORD;
97              break;
98            case EST:
99              dir2 = Stanza.Direzioni.OVEST;
100             break;
101           case OVEST:
102             dir2 = Stanza.Direzioni.EST;
103             break;
104           }
105           int dst = tas.nextInt();
106           if ((dst < 0) || (dst > MAX_STANZE))
107             throw new ErroreMappaException(
108               fmappa + ": numero " + dst + " fuori range");
```

```
109           if (stanze[dst] == null)
110             throw new ErroreMappaException(
111               fstanze + ": stanza " + dst + " non esiste");
112           Porta p = new Porta(stanze[src], stanze[dst]);
113           stanze[src].collega(dir1, p);
114           stanze[dst].collega(dir2, p);
115         }
116       } catch (Exception e) {}
117       finally {
118         try {
119           br.close();
120         } catch (IOException e) {}
121       }
122       return init;
123     }
124   }
```

Test.java:

```
1   package cap11.mappalabirinto;
2
3   import IngressoUscita.Console;
4   import cap6.labirinto.*;
5   class Test {
6     public static void main(String[] args) {
7       String fstanze = "stanze.txt";
8       String fmappa = "mappa.txt";
9       int n = (args.length < 2) ? args.length : 2;
10      switch (n) {
11      case 2:
12        fmappa = args[1];
13      case 1:
14        fstanze = args[0];
15      }
16      Mappa m = new Mappa(fstanze, fmappa);
17      Giocatore giocatore = new Giocatore("Zak");
18      Stanza s;
19      try {
20        s = m.creaLabirinto();
21      } catch (Exception e) {
22        Console.scriviStringa(e.getMessage());
23        return;
```

```
24        }
25        Console.scriviStringa("Benvenuto.");
26        giocatore.muovi(s);
27        boolean ancora = true;
28        while (ancora) {
29          s = giocatore.locazione();
30          Console.scriviStringa(s.descrivi());
31          Console.scriviStringa("Dove vuoi andare?");
32          String dir = Console.leggiStringa();
33          if (dir.equals("fine"))
34            ancora = false;
35          else
36            s.vai(
37              giocatore,
38              Stanza.Direzioni.valueOf(dir.toUpperCase()));
39        }
40      }
41    }
```

12. Letture e scritture di oggetti

12.1 Tavola

Una tavola è formata da caselle disposte su N righe e N colonne. Le righe e le colonne della tavola sono numerate a partire da 0. Ciascuna casella può essere libera oppure contrassegnata con un simbolo, ed i possibili simboli sono un cerchio ed una croce. Ciascun simbolo rappresenta un giocatore. I giocatori a turno contrassegnano mediante il proprio simbolo una casella libera. Le operazioni che possono essere effettuate su una tavola sono elencate di seguito.

- *Tavola()*: costruttore di default, che inizializza una tavola formata da 3 righe e 3 colonne. Inizialmente tutte le caselle della tavola sono libere ed il turno è del giocatore avente per simbolo il cerchio.
- *Tavola(int n)*: costruttore che inizializza una tavola formata da n righe e n colonne. Inizialmente tutte le caselle della tavola sono libere ed il turno è del giocatore avente per simbolo il cerchio.
- *void contrassegna(int r, int c)*: operazione che contrassegna la casella di riga r e colonna c. La casella viene contrassegnata con il simbolo del giocatore di turno, ed il turno passa all'altro giocatore. Il metodo non esegue nessuna azione se la casella di riga r e colonna c non è libera.
- *boolean cerchio(int r, int c)*: restituisce *true* se la casella di riga r e colonna c è contrassegnata da un cerchio, *false* altrimenti.
- *boolean croce(int r, int c)*: restituisce *true* se la casella di riga r e colonna c è contrassegnata da una croce, *false* altrimenti.

- *boolean vinceCerchio()*: restituisce *true* se la tavola contiene almeno una sequenza di *N* cerchi disposti lungo una riga, una colonna, od una diagonale (*false* altrimenti).
- *boolean vinceCroce()*: restituisce *true* se la tavola contiene almeno una sequenza di *N* croci disposte lungo una riga, una colonna, od una diagonale (*false* altrimenti).
- *boolean piena()*: restituisce *true* se la tavola è piena, *false* altrimenti.
- *void svuota()*: riporta la tavola nello stato iniziale.
- *void salva(String f) throws IOException*: salva lo stato della tavola, in formato binario, nel file che ha nome *f*.
- *static Tavola carica(String f) throws IOException, ClassNotFoundException*: restituisce un oggetto *Tavola* caricandone lo stato dal file con nome *f*.
- *String toString()*: restituisce una stringa del tipo:

```
O - X
X X O
O - -
```

Soluzione

Tavola.java:

```
1  package cap12.tavola;
2
3  import java.io.*;
4  public class Tavola implements Serializable {
5    public static enum Stato {
6      LIBERA("-"), CRO("X"), CER("O");
7      private String s;
8      private Stato(String a) {
9        s = a;
10     }
11     public String toString() {
12       return s;
13     }
14   }
15   private static enum Giocatore {
16     CERCHIO("Giocatore O"), CROCE("Giocatore X");
```

```
17      private String nome;
18      private Giocatore(String s) {
19        nome = s;
20      }
21      public String toString() {
22        return nome;
23      }
24    }
25    private Stato[][] tav;
26    private Giocatore gio;
27    private final int N;
28    public Tavola() {
29      this(3);
30    }
31    public Tavola(int n) {
32      N = n;
33      tav = new Stato[N][N];
34      for (int i = 0; i < N; i++)
35        for (int j = 0; j < N; j++)
36          tav[i][j] = Stato.LIBERA;
37      gio = Giocatore.CERCHIO;
38    }
39    public boolean cerchio(int r, int c) {
40      return tav[r][c] == Stato.CER;
41    }
42    public boolean croce(int r, int c) {
43      return tav[r][c] == Stato.CRO;
44    }
45    public void contrassegna(int r, int c) {
46      if (tav[r][c] != Stato.LIBERA)
47        return;
48      tav[r][c] = ((gio == Giocatore.CERCHIO) ? Stato.CER
49                   : Stato.CRO);
50      gio = ((gio == Giocatore.CERCHIO) ? Giocatore.CROCE
51             : Giocatore.CERCHIO);
52    }
53    public void svuota() {
54      for (int i = 0; i < N; i++)
55        for (int j = 0; j < N; j++)
56          tav[i][j] = Stato.LIBERA;
57      gio = Giocatore.CERCHIO;
```

```
58    }
59    private boolean cerca(Stato st) {
60      int i;
61      int j;
62      for (i = 0; i < N; i++) {
63        for (j = 0; (j < N) && (tav[i][j] == st); j++)
64          ;
65        if (j == N)
66          return true;
67      }
68      for (j = 0; j < N; j++) {
69        for (i = 0; (i < N) && (tav[i][j] == st); i++)
70          ;
71        if (i == N)
72          return true;
73      }
74      for (i = 0; (i < N) && (tav[i][i] == st); i++)
75        ;
76      if (i == N)
77        return true;
78      for (i = 0; (i < N) && (tav[i][N - i - 1] == st);
79           i++)
80        ;
81      if (i == N)
82        return true;
83      return false;
84    }
85    public boolean vinceCerchio() {
86      return cerca(Stato.CER);
87    }
88    public boolean vinceCroce() {
89      return cerca(Stato.CRO);
90    }
91    public boolean piena() {
92      for (int i = 0; i < N; i++)
93        for (int j = 0; j < N; j++)
94          if (tav[i][j] == Stato.LIBERA)
95            return false;
96      return true;
97    }
98    public Stato elemento(int r, int c) {
```

```
99        return tav[r][c];
100     }
101     public String toString() {
102       String s = "";
103       for (int i = 0; i < N; i++) {
104         for (int j = 0; j < N; j++)
105           s += tav[i][j];
106         s += System.getProperty("line.separator");
107       }
108       return s;
109     }
110     public void salva(String s) throws IOException {
111       ObjectOutputStream oos =
112         new ObjectOutputStream(new FileOutputStream(s));
113       oos.writeObject(this);
114       oos.close();
115     }
116     public static Tavola carica(String s)
117     throws IOException, ClassNotFoundException {
118       ObjectInputStream ois =
119         new ObjectInputStream(new FileInputStream(s));
120       return (Tavola) ois.readObject();
121     }
122   }
```

Note

Per realizzare i metodi *salva()* e *carica()*, è sufficiente dichiarare che la classe *Tavola* implementa l'interfaccia *Serializable* (linea 5), e usare gli stream di tipo *ObjectOutputStream* (linea 113) e *ObjectInputStream* (linea 120).

Notare gli enumerati *Stato* (linee 7–14) e *Giocatore* (linee 17–24), che sfruttano la possibilità di avere dei campi e dei metodi per memorizzare al loro interno la stringa che li rappresenta.

La funzione *cerca()* è strutturata nel seguente modo. Linee 63–68: cerca una riga qualunque che contenga *N* simboli uguali a *st*. Linee 69–74: cerca una colonna qualunque che contenga *N* simboli uguali a *st*. Linee 75–78: controlla se la diagonale discendente è composta solo da simboli uguali a *st*. Linee 79–83: controlla se la diagonale ascendente è composta solo da simboli uguali a *st*. Linea 84: se nessuna delle precedenti ricerche ha avuto successo, restituisce *false*.

12.2 Salva gioco

Senza modificare le classi definite nell'esercizio 6.11, fare in modo che sia possibile salvare e caricare lo stato del gioco (stanze e porte del labirinto comprese). Scrivere quindi un nuovo programma di test che, oltre alle direzioni e al comando "fine", comprende anche il comando "salva". Tale comando deve salvare lo stato del gioco in un file "labirinto.dat". All'avvio, il programma deve prima provare a caricare lo stato del gioco da tale file e solo se ciò non è possibile (per esempio perché il file non esiste) deve creare un nuovo labirinto come nel programma di test dell'esercizio 6.11. Si può assumere che il labirinto è completamente connesso.

Suggerimenti

Poiché le classi definite in 6.11 non implementano l'interfaccia *Serializable*, la soluzione dell'esercizio richiede la definizione di una opportuna sottoclasse per ognuna delle classi rilevanti, e l'uso dell'interfaccia *Externalizable*.

Soluzione

GiocatoreSalva.java:

```
1  package cap12.salvagioco;
2
3  import cap6.labirinto.*;
4  import java.io.*;
5  public class GiocatoreSalva extends Giocatore
6    implements Externalizable {
7    public GiocatoreSalva() {
8      super("");
9    }
10   public GiocatoreSalva(String nome) {
11     super(nome);
12   }
13   public void readExternal(ObjectInput in)
14   throws IOException, ClassNotFoundException {
15     stanza = (Stanza) in.readObject();
16     nome = (String) in.readObject();
17   }
18   public void writeExternal(ObjectOutput out)
19   throws IOException {
20     out.writeObject(stanza);
21     out.writeObject(nome);
```

```
22    }
23  }
```

PortaSalva.java:

```
1   package cap12.salvagioco;
2
3   import cap6.labirinto.*;
4   import java.io.*;
5   public class PortaSalva extends Porta
6     implements Externalizable {
7     public PortaSalva() {
8       super(null, null);
9     }
10    public PortaSalva(Stanza stanza1, Stanza stanza2) {
11      super(stanza1, stanza2);
12    }
13    public void readExternal(ObjectInput in)
14    throws IOException, ClassNotFoundException {
15      stanza1 = (Stanza) in.readObject();
16      stanza2 = (Stanza) in.readObject();
17    }
18    public void writeExternal(ObjectOutput out)
19    throws IOException {
20      out.writeObject(stanza1);
21      out.writeObject(stanza2);
22    }
23  }
```

StanzaSalva.java:

```
1   package cap12.salvagioco;
2
3   import cap6.labirinto.*;
4   import java.io.*;
5   public class StanzaSalva extends Stanza
6     implements Externalizable {
7     public StanzaSalva() {
8       super("");
9     }
10    public StanzaSalva(String descr) {
11      super(descr);
```

```
12      }
13      public void readExternal(ObjectInput in)
14      throws IOException, ClassNotFoundException {
15        descr = (String) in.readObject();
16        porte = (Porta[]) in.readObject();
17      }
18      public void writeExternal(ObjectOutput out)
19      throws IOException {
20        out.writeObject(descr);
21        out.writeObject(porte);
22      }
23    }
```

Note

Per salvare tutto lo stato del gioco è sufficiente salvare l'oggetto *Giocatore*. Infatti, poiché il salvataggio è ricorsivo, attraverso il campo *stanza* della classe *Giocatore* verrà salvato l'oggetto *Stanza* corrispondente alla stanza in cui il giocatore si trova, quindi, attraverso l'array *porte* della classe *Stanza*, verranno salvati gli oggetti *Porta* collegati a quella stanza, e così via. Poiché abbiamo supposto che il labirinto fosse completamente connesso, tutto gli oggetti *Stanza* e *Porta* verranno salvati.

Si noti la necessità di definire un costruttore di default per le classi *GiocatoreSalva*, *PortaSalva* e *StanzaSalva*. Tale costruttore viene chiamato dal metodo *readObject*, prima di caricare i campi dell'oggetto dal file.

13. Generici

13.1 Coda

Realizzare, tramite una lista, una classe generica *CodaSemplice* che implementi la seguente interfaccia:

Coda.java:

```
1  package cap13.coda;
2
3  public interface Coda<T> {
4    boolean vuota();
5    void in(T info);
6    T out() throws CodaVuotaException;
7  }
```

L'interfaccia descrive una coda FIFO in cui il metodo *in()* esegue l'inserimento in coda e il metodo *out()* l'estrazione dalla testa.

Soluzione

CodaVuotaException.java:

```
1  package cap13.coda;
2
3  public class CodaVuotaException extends Exception {
4  }
```

CodaSemplice.java:

```
1  package cap13.coda;
2
3  public class CodaSemplice<T> implements Coda<T> {
4    private class Elem<A> {
5      A info;
6      Elem<A> next;
7      Elem(A info) {
8        this.info = info;
9      }
10   };
11
12   private Elem<T> testa = null;
13   private Elem<T> coda = null;
14   public boolean vuota() {
15     return testa == null;
16   }
17   public void in(T info) {
18     Elem<T> tmp = new Elem<T>(info);
19     if (coda != null)
20       coda.next = tmp;
21     coda = tmp;
22     if (testa == null)
23       testa = coda;
24   }
25   public T out() throws CodaVuotaException {
26     if (vuota())
27       throw new CodaVuotaException();
28     Elem<T> tmp = testa;
29     testa = testa.next;
30     if (testa == null)
31       coda = null;
32     return tmp.info;
33   }
34 }
```

13.2 Insieme

Vogliamo scrivere una classe che realizzi il concetto matematico di insieme: una collezione di elementi in cui non ci sono duplicati. In particolare vogliamo che *i*) non ci siano all'interno dell'insieme due elementi *e1* ed *e2* tali che *e1.equals(e2)*

restituisca *true*, *ii*) il valore *null* possa essere contenuto in un insieme al più una volta.

La classe deve fornire almeno i seguenti metodi (il tipo *T* è un tipo generico):

- *int quanti()*: restituisce il numero di elementi contenuti nell'insieme.
- *boolean vuoto()*: restituisce *true* se applicato ad un insieme vuoto, *false* altrimenti.
- *Insieme<T> copia()*: restituisce un nuovo oggetto di tipo *Insieme* che contiene gli stessi elementi dell'oggetto a cui viene applicato.
- *boolean contiene(T t)*: restituisce *true* se l'insieme contiene l'elemento T, *false* altrimenti.
- *boolean aggiungi(T t)*: aggiunge, se non già presente, l'elemento *t* all'insieme; restituisce *true* se l'inserimento avviene con successo, *false* altrimenti.
- *boolean rimuovi(T t)*: rimuove l'elemento *t* dall'insieme; restituisce *true* se l'elemento da rimuovere era presente, *false* altrimenti.
- *Iterator<T> iterator()*: restituisce un iteratore che può essere usato per scorrere e rimuovere gli elementi dell'insieme.
- *Insieme<T> intersezione(Insieme<T> it)*: restituisce un nuovo insieme pari all'intersezione dell'oggetto implicito e di *it* (l'intersezione contiene gli elementi a comune tra i due insiemi).
- *Insieme<T> unione(Insieme<T> it)*: restituisce un nuovo insieme pari all'unione dell'oggetto implicito e di *it* (l'unione contiene gli elementi che appartengono a uno qualunque dei due insiemi).
- *Insieme<T> differenza(Insieme<T> it)*: restituisce un nuovo insieme pari alla differenza tra l'oggetto implicito e *it* (gli elementi dell'oggetto implicito che non sono contenuti in *it*).

L'interfaccia *Iterator<E>* fa parte del package *java.util* e prevede i seguenti metodi:

- *boolean hasNext()*: restituisce *true* se ci sono ancora elementi da visitare nell'iteratore, *false* altrimenti.
- *E next()*: restituisce il prossimo elemento dell'iteratore. Lancia una istanza dell'eccezione *NoSuchElementException* (una eccezione di tipo *unchecked* del package *java.util*) se non ci sono più elementi da restituire.

- *void remove()*: rimuove dal "contenitore" associato all'iteratore l'ultimo oggetto restituito dall'iteratore. Questo metodo può essere invocato una sola volta per ogni chiamata al metodo *next()*. Lancia una eccezione di tipo *IllegalStateException* (di tipo *unchecked* e appartenente al package *java.lang*) se il metodo *next()* non è stato mai chiamato o se il metodo *remove()* era stato già chiamato dopo l'ultima invocazione del metodo *next()*. Gli iteratori che non vogliono implementare la funzione di rimozione di un elemento devono lanciare una eccezione di tipo *UnsupportedOperationException* (di tipo *unchecked* e appartenente al package *java.lang*) quando il metodo *remove()* viene invocato.

Soluzione

Insieme.java:

```
1  package cap13.insieme;
2
3  import java.util.Iterator;
4  public abstract class Insieme<T> {
5    public abstract int quanti();
6    public abstract boolean vuoto();
7    public abstract boolean contiene(T t);
8    public abstract boolean aggiungi(T t);
9    public abstract boolean rimuovi(T t);
10   public abstract Insieme<T> copia();
11   public abstract Iterator<T> iterator();
12   public Insieme<T> unione(Insieme<T> it) {
13     Insieme<T> rr = copia();
14     Iterator<T> ii = it.iterator();
15     while (ii.hasNext())
16       rr.aggiungi(ii.next());
17     return rr;
18   }
19   public Insieme<T> intersezione(Insieme<T> it) {
20     Insieme<T> rr = copia();
21     Iterator<T> ii = rr.iterator();
22     while (ii.hasNext()) {
23       T t = ii.next();
24       if (!it.contiene(t))
25         rr.rimuovi(t);
26     }
27     return rr;
```

```
28    }
29    public Insieme<T> differenza(Insieme<T> it) {
30      Insieme<T> rr = copia();
31      Iterator<T> ii = it.iterator();
32      while (ii.hasNext())
33        rr.rimuovi(ii.next());
34      return rr;
35    }
36    public String toString() {
37      String rr = "{";
38      boolean primo = true;
39      Iterator<T> ii = iterator();
40      while (ii.hasNext()) {
41        T t = ii.next();
42        if (!primo)
43          rr += ", ";
44        else
45          primo = false;
46        rr += ((t == null) ? "null" : t);
47      }
48      return rr + "}";
49    }
50  }
```

InsiemeList.java:

```
1   package cap13.insieme;
2
3   import java.util.Iterator;
4   public class InsiemeList<T> extends Insieme<T> {
5     private static class Elemento<A> {
6       A info;
7       Elemento<A> next;
8       boolean uguale(A a) {
9         return ((a == null) && (info == null)) ||
10                ((a != null) && (info != null) && info.equals(a));
11      }
12    }
13    Elemento<T> testa;
14    private int n;
15    public int quanti() {
16      return n;
```

```
17      }
18      public boolean vuoto() {
19        return n == 0;
20      }
21      public Insieme<T> copia() {
22        Insieme<T> rr = new InsiemeList<T>();
23        Iterator<T> ii = iterator();
24        while (ii.hasNext())
25          rr.aggiungi(ii.next());
26        return rr;
27      }
28      public boolean contiene(T t) {
29        Elemento<T> e = testa;
30        while (e != null)
31          if (e.uguale(t))
32            return true;
33          else
34            e = e.next;
35        return false;
36      }
37      public boolean aggiungi(T t) {
38        if (contiene(t))
39          return false;
40        Elemento<T> tmp = new Elemento<T>();
41        tmp.info = t;
42        tmp.next = testa;
43        testa = tmp;
44        n++;
45        return true;
46      }
47      public boolean rimuovi(T t) {
48        if (testa == null)
49          return false;
50        if (testa.uguale(t)) {
51          testa = testa.next;
52          n--;
53          return true;
54        }
55        Elemento<T> p = testa;
56        Elemento<T> q = testa.next;
57        while (q != null)
```

```
58          if (q.uguale(t)) {
59            p.next = q.next;
60            n--;
61            return true;
62          } else {
63            p = q;
64            q = q.next;
65          }
66        return false;
67      }
68      public Iterator<T> iterator() {
69        return new InsiemeListaIteratore();
70      }
71      class InsiemeListaIteratore implements Iterator<T> {
72        Elemento<T> r;
73        Elemento<T> p;
74        boolean ancora;
75        boolean rem;
76        InsiemeListaIteratore() {
77          ancora = quanti() != 0;
78          rem = true;
79        }
80        public boolean hasNext() {
81          return ancora;
82        }
83        public T next() {
84          if (!ancora)
85            throw new IllegalStateException();
86          p = r;
87          if (r == null)
88            r = testa;
89          else
90            r = r.next;
91          ancora = r.next != null;
92          rem = false;
93          return r.info;
94        }
95        public void remove() {
96          if (rem)
97            throw new IllegalStateException();
98          if (r == testa)
```

```
 99            testa = testa.next;
100          else
101            p.next = r.next;
102          n--;
103          rem = true;
104        }
105      }
106    }
```

Note

La soluzione proposta si compone di due classi: *Insieme* e *InsiemeList*. La prima è una classe astratta che contiene il codice comune a tutte le eventuali realizzazioni del concetto di insieme, mentre la seconda fornisce una implementazione a lista. In questo modo una eventuale realizzazione alternativa, per esempio basata su array, potrebbe essere scritta con poco sforzo riutilizzando tutto il codice della superclasse astratta.

Per usare tali classi il programmatore può usare istruzioni quali le seguenti:

```
Insieme<String> i1 = new InsiemeList<String>();
if(i1.aggiungi("uno"))
  ...
```

che creano un insieme di stringhe in cui è presente un solo elemento. Quindi, sull'oggetto *i1* possono essere invocati tutti i metodi della classe *Insieme*. Se fosse disponibile anche una realizzazione basata su array, il programmatore potrebbe anche scrivere:

```
Insieme<String> i2 = new InsiemeArray<String>();
...
Insieme<String> i3 = i2.unione(i1);
...
```

14. Thread

14.1 Thread semplici

Si scrivano le seguenti classi:

- *ContoThread* dotata di un costruttore che prende come parametro un intero *n*; quando il thread viene attivato, viene stampato a video un conto alla rovescia che va da *n* a *0*, con cadenza pari ad un secondo.

- *FiboThread* dotata di un costruttore che prende come parametro un intero *n*; quando il thread viene attivato vengono stampati a video, con la cadenza di uno al secondo, i termini della successione di Fibonacci, partendo dal primo e fino ad arrivare al termine *n*-esimo.

- *Prova* una classe dotata di un metodo *main()* che stampa a video un semplice menu simile al seguente:

```
Quale tipo di thread vuoi lanciare?
1) conto alla rovescia
2) fibonacci
3) tutti e due insieme
1
Inserisci il valore di n: 10
10
9
8
...
```

Soluzione

ContoThread.java:

```
1  package cap14.semplici;
2  import IngressoUscita.*;
3  public class ContoThread extends Thread {
4    int n;
5    public ContoThread(int n) {
6      this.n = n;
7    }
8    public void run() {
9      for(; n>=0; n--) {
10       try {
11         Thread.sleep(1000);
12       } catch(InterruptedException ie) {
13         Console.scriviStringa("Thread interrotto");
14       }
15       Console.scriviIntero(n);
16     }
17   }
18 }
```

FiboThread.java:

```
1  package cap14.semplici;
2
3  import IngressoUscita.*;
4  public class FiboThread extends Thread {
5    int n;
6    public FiboThread(int n) {
7      this.n = n;
8    }
9    public void run() {
10     int i1 = 0;
11     int i2 = 1;
12     for (; n > 0; n--) {
13       try {
14         Thread.sleep(1000);
15         int tmp = i1;
16         i1 = i2;
17         i2 += tmp;
18       } catch (InterruptedException ie) {
```

```
19            Console.scriviStringa("Thread interrotto");
20          }
21          Console.scriviIntero(i2);
22        }
23      }
24    }
```

Prova.java:

```
1   package cap14.semplici;
2   import IngressoUscita.*;
3   class Prova {
4     public static void main(String[] args) {
5       Console.scriviStringa("Quale tipo di thread vuoi lanciare?");
6       Console.scriviStringa("1) conto alla rovescia");
7       Console.scriviStringa("2) fibonacci");
8       Console.scriviStringa("3) tutti e due insieme");
9       int s = Console.leggiIntero();
10      Console.scriviStringa("n?");
11      int n = Console.leggiIntero();
12      Thread t1, t2;
13      switch(s) {
14      case 1:
15        t1 = new ContoThread(n);
16        t1.start();
17        break;
18      case 2:
19        t2 = new FiboThread(n);
20        t2.start();
21        break;
22      case 3:
23        t1 = new ContoThread(n);
24        t1.start();
25        t2 = new FiboThread(n);
26        t2.start();
27        break;
28      default:
29        Console.scriviStringa("Scelta non valida");
30      }
31    }
32  }
```

14.2 Thread interrotti

Si modifichi l'esercizio 14.1 in modo che il metodo *main()* della classe *Prova* interrompa i thread *t* secondi, dove il valore di *t* è chiesto all'utente di prima di avviare i thread.

Soluzione

ContoThread.java:

```
1  package cap14.semplici;
2  import IngressoUscita.*;
3  public class ContoThread extends Thread {
4    int n;
5    public ContoThread(int n) {
6      this.n = n;
7    }
8    public void run() {
9      for(; n>=0; n--) {
10       try {
11         Thread.sleep(1000);
12       } catch(InterruptedException ie) {
13         Console.scriviStringa("Thread interrotto");
14       }
15       Console.scriviIntero(n);
16     }
17   }
18 }
```

FiboThread.java:

```
1  package cap14.semplici;
2
3  import IngressoUscita.*;
4  public class FiboThread extends Thread {
5    int n;
6    public FiboThread(int n) {
7      this.n = n;
8    }
9    public void run() {
10     int i1 = 0;
11     int i2 = 1;
```

```
12      for (; n > 0; n--) {
13        try {
14          Thread.sleep(1000);
15          int tmp = i1;
16          i1 = i2;
17          i2 += tmp;
18        } catch (InterruptedException ie) {
19          Console.scriviStringa("Thread interrotto");
20        }
21        Console.scriviIntero(i2);
22      }
23    }
24  }
```

Prova.java:

```
1   package cap14.semplici;
2   import IngressoUscita.*;
3   class Prova {
4     public static void main(String[] args) {
5       Console.scriviStringa("Quale tipo di thread vuoi lanciare?");
6       Console.scriviStringa("1) conto alla rovescia");
7       Console.scriviStringa("2) fibonacci");
8       Console.scriviStringa("3) tutti e due insieme");
9       int s = Console.leggiIntero();
10      Console.scriviStringa("n?");
11      int n = Console.leggiIntero();
12      Thread t1, t2;
13      switch(s) {
14      case 1:
15        t1 = new ContoThread(n);
16        t1.start();
17        break;
18      case 2:
19        t2 = new FiboThread(n);
20        t2.start();
21        break;
22      case 3:
23        t1 = new ContoThread(n);
24        t1.start();
25        t2 = new FiboThread(n);
26        t2.start();
```

```
27          break;
28        default:
29          Console.scriviStringa("Scelta non valida");
30        }
31      }
32    }
```

14.3 Versa e preleva

Si scrivano le seguenti classi:

- *Conto* con un campo privato *bilancio* e metodi pubblici *void versa(int somma)*, *void preleva(int somma)*, e *String toString()* (quest'ultimo restituisce il valore di bilancio sotto forma di stringa); non è possibile prelevare da un conto una somma maggiore di quella disponibile.
- *VersaThread* che estende la classe *Thread* ed è dotata di un costruttore che prende come argomenti un oggetto *Conto c* e un intero *s*; quando viene avviato, il thread deve versare la somma *s* sul conto *c*, quindi stampare la scritta "Ho versato" e terminare.
- *PrelevaThread* che estende la classe *Thread* ed è dotata di un costruttore che prende come argomenti un oggetto *Conto c* e un intero *s*; quando viene avviato, il thread deve prelevare la somma *s* dal conto *c* e , una volta avvenuto il prelievo, stampare la scritta "Ho prelevato" e terminare.

Realizzare infine una classe di prova che crea un oggetto *Conto*, un thread di tipo *VersaThread* e *PrelevaThread* e li avvia.

Soluzione

Conto.java:

```
1   package cap14.threadbanca;
2
3   public class Conto {
4     private int bilancio;
5
6     public synchronized void versa(int somma) {
7       bilancio += somma;
8       notify();
9     }
```

```
10
11    public synchronized void preleva(int somma) throws
12      InterruptedException {
13      while ((bilancio - somma) < 0)
14        wait();
15      bilancio -= somma;
16    }
17
18    public String toString() {
19      return String.valueOf(bilancio);
20    }
21  }
```

VersaThread.java:

```
1   package cap14.threadbanca;
2
3   public class VersaThread extends Thread {
4     private int s;
5     private Conto c;
6     public VersaThread(Conto c, int s) {
7       this.c = c;
8       this.s = s;
9     }
10    public void run() {
11      c.versa(s);
12      System.out.println("Ho versato");
13    }
14  }
```

PrelevaThread.java:

```
1   package cap14.threadbanca;
2
3   public class PrelevaThread extends Thread {
4     private int s;
5     private Conto c;
6
7     public PrelevaThread(Conto c, int s) {
8       this.c = c;
9       this.s = s;
10    }
```

```
11
12    public void run() {
13      try {
14        c.preleva(s);
15        System.out.println("Ho prelevato");
16      } catch (InterruptedException ie) {
17        System.out.println("Sono stato interrotto");
18      }
19    }
20  }
```

Prova.java:

```
1   package cap14.threadbanca;
2
3   public class Prova {
4     public static void main(String[] args) {
5       Conto c = new Conto();
6       PrelevaThread pt = new PrelevaThread(c, 100);
7       VersaThread vt = new VersaThread(c, 150);
8       pt.start();
9       vt.start();
10    }
11  }
```

14.4 Dadi

Il valore di un dado, compreso tra uno e sei, varia in continuazione quando il dado rotola, e rimane fisso quando il dado è fermo. Quando un dado rotola assume un nuovo valore casuale ogni p millisecondi. Un dado può essere rappresentato da una istanza di una classe *Dado*, estensione della classe *Thread*, dotata almeno dei seguenti costruttori e metodi:

- *Dado()*: crea un dado con un valore di p pari a 500 millisecondi.
- *Dado(int p)*: crea un dado con il valore di p specificato.
- *void vai()*: il dado comincia a rotolare variando il suo valore ogni p millisecondi.
- *void ferma()*: ferma temporaneamente il dado (una nuova chiamata al metodo *vai()* fa ripartire il dado).

- *void fine()*: ferma definitivamente il dado.
- *int dammiValore()*: restituisce il valore corrente del dado.

Un lancio è composto da più dadi. Definire una classe *Lancio* con le seguenti caratteristiche:

- *Lancio(int n)*: crea un lancio composto da *n* dadi con il valore predefinito di *p*.
- *Lancio(int n, int p)*: crea un lancio composto da *n* dadi con il valore di *p* specificato.
- *void vai()*: tutti i dadi cominciano a rotolare.
- *void ferma()*: ferma temporaneamente tutti i dadi (una nuova chiamata al metodo *vai()* fa ripartire i dadi).
- *void fine()*: ferma definitivamente i dadi.
- *String toString()*: restituisce una stringa che rappresenta il valore corrente dei vari dadi, per esempio `< 2 4 1 >` (il lancio è composto da tre dadi, che nell'istante in cui viene chiamato il metodo *toString()* hanno i valori 2, 4 e 1).

Scrivere infine un programma di prova che: *i*) crea un lancio di dadi e li comincia a far rotolare; *ii*) ne stampa periodicamente il valore per un certo numero di volte; *iii*) ferma i dadi prima di terminare.

Soluzione

Dado.java:

```
1  package cap14.dadi;
2
3  public class Dado extends Thread {
4    private static final int VALORE_MAX = 6;
5    private static final int VALORE_MIN = 1;
6    private static final int PERIODO_DEFAULT = 500;
7    private int valore;
8    private boolean paused = true;
9    private int periodo = PERIODO_DEFAULT;
10
11   public Dado() {
12     start();
```

```
13      }
14
15      public Dado(int p) {
16        periodo = p;
17        start();
18      }
19
20      public synchronized void vai() {
21        paused = false;
22        notify();
23      }
24
25      public synchronized void ferma() {
26        paused = true;
27      }
28
29      public void fine() {
30        interrupt();
31      }
32
33      public synchronized int dammiValore() {
34        return valore;
35      }
36
37      private synchronized void controlla() throws
38        InterruptedException {
39        if (paused) {
40          wait();
41        }
42      }
43
44      public String toString() {
45        return Integer.toString(dammiValore());
46      }
47
48      public void run() {
49        try {
50          while (!isInterrupted()) {
51            controlla();
52            valore = 1 + (int) (Math.random() * (VALORE_MAX -
53                                                 VALORE_MIN
```

```
54                                                  + 1));
55            sleep(periodo);
56          }
57        } catch (InterruptedException e) {
58        }
59      }
60    }
```

Lancio.java:

```
1   package cap14.dadi;
2
3   public class Lancio {
4     private Dado[] dad;
5     public Lancio(int n, int p) {
6       dad = new Dado[n];
7       for (int i = 0; i < n; i++)
8         dad[i] = new Dado(p);
9     }
10    public Lancio(int n) {
11      dad = new Dado[n];
12      for (int i = 0; i < n; i++)
13        dad[i] = new Dado();
14    }
15    public void vai() {
16      for (int i = 0; i < dad.length; i++)
17        dad[i].vai();
18    }
19    public void ferma() {
20      for (int i = 0; i < dad.length; i++)
21        dad[i].ferma();
22    }
23    public void fine() {
24      for (int i = 0; i < dad.length; i++)
25        dad[i].fine();
26    }
27    public String toString() {
28      String s = "< ";
29      for (int i = 0; i < dad.length; i++)
30        s += (dad[i] + " ");
31      s += ">";
32      return s;
```

```
33      }
34    }
```

ProvaLancio.java:

```
1   package cap14.dadi;
2
3   import IngressoUscita.Console;
4   public class ProvaLancio {
5     public static void main(String[] args) {
6       Console.scriviStringa("Quanti dadi ?");
7       int n = Console.leggiIntero();
8       Console.scriviStringa("Inserisci il valore di p");
9       int p = Console.leggiIntero();
10      Console.scriviStringa(
11        "Quante volte ne vuoi stampare il valore ?");
12      int q = Console.leggiIntero();
13      Lancio la = new Lancio(n, p);
14      la.vai();
15      for (int i = 0; i < q; i++) {
16        try {
17          Thread.sleep(p);
18        } catch (InterruptedException e) {}
19        Console.scriviStringa(la.toString());
20      }
21      la.ferma();
22      Console.scriviStringa(la.toString());
23      la.fine();
24    }
25  }
```

14.5 Ladro

Vogliamo estendere il gioco introdotto nell'esercizio 9.8 (o quello introdotto nell'esercizio 9.9) nel seguente modo: nel labirinto si aggira un ladro, che può rubare gli oggetti che trova nel suo cammino e portarli nel suo rifugio, all'interno del labirinto stesso.

Per realizzare questa estensione, facciamo in modo che sia il giocatore, sia il ladro, utilizzino istanze della classe *Giocatore* e introduciamo due thread: un thread muove il giocatore (secondo i comandi impartiti dalla tastiera) e l'altro muove il ladro, a caso.

Aggiungere alle classi dell'esercizio 9.8 (o 9.9) le classi *RunGiocatore* e *RunLadro* che estendono la classe *Thread* e contengono almeno i metodi elencati di seguito.

Classe *RunGiocatore*:

- *RunGiocatore(Giocatore g)*: inizializza un nuovo oggetto *RunGiocatore*, associato all'oggetto *Giocatore g*.
- *run()*: ciclicamente, mostra sulla console la descrizione della stanza in cui il giocatore si trova, quindi legge una stringa da tastiera e la interpreta opportunamente. L'interpretazione deve riconoscere almeno i comandi "`vai` *direzione*", "`prendi` *oggetto*", "`lascia` *oggetto*", "`inventario`" e "`fine`".

Classe *RunLadro*:

- *RunLadro(Giocatore ladro, Stanza rifugio)*: inizializza un nuovo oggetto *RunLadro*, associato al *Giocatore* ladro, il cui rifugio è la stanza *rifugio*.
- *run()*: se il ladro si trova nel suo rifugio, lascia tutti gli oggetti eventualmente trasportati; se il ladro non si trova nel suo rifugio, prende gli oggetti che, eventualmente, si trovano nella stanza in cui si trova (per ogni oggetto presente, la probabilità che il ladro lo prenda deve essere del 70%); in ogni caso, prova a spostarsi in una direzione a caso, quindi si ferma per 3 secondi e riparte da capo.

Suggerimenti

Per realizzare le scelte casuali del ladro, si può usare la stessa tecnica usata nell'esercizio 10.3.

Notare che sia il giocatore, sia il ladro, potrebbero cercare di prendere o lasciare oggetti nella stessa stanza, o cercare di prendere lo stesso oggetto, contemporaneamente. È allora necessario che le operazioni di *Stanza* e *MobileImpl* siano sincronizzate. Siccome, però, non vogliamo modificare le classi *Stanza* e *MobileImpl* già scritte, è consigliabile creare due nuove classi *StanzaSyncr* e *MobileImplSyncr* che estendono le precedenti....

Soluzione

StanzaSyncr.java:

```
1  package cap14.ladro;
2
3  import cap9.prenderelasciare.*;
4  import cap6.labirinto.Risultato;
```

```
5  public class StanzaSyncr extends Stanza {
6    public StanzaSyncr(String descr) {
7      super(descr);
8    }
9    public synchronized Oggetto[] elenco() {
10     return super.elenco();
11   }
12   public synchronized Risultato aggiungi(Oggetto o) {
13     return super.aggiungi(o);
14   }
15   public synchronized Risultato rimuovi(Oggetto o) {
16     return super.rimuovi(o);
17   }
18 }
```

MobileImplSyncr.java:

```
1  package cap14.ladro;
2
3  import cap6.labirinto.Risultato;
4  import cap9.prenderelasciare.*;
5  public class MobileImplSyncr extends MobileImpl {
6    public MobileImplSyncr(String nome,
7                           Contenitore luogo) {
8      super(nome, luogo);
9    }
10   public synchronized Contenitore luogo() {
11     return super.luogo();
12   }
13   public synchronized Risultato sposta(Contenitore sorg,
14                                        Contenitore dest) {
15     return super.sposta(sorg, dest);
16   }
17 }
```

RunGiocatore.java:

```
1  package cap14.ladro;
2
3  import IngressoUscita.Console;
4  import cap9.prenderelasciare.*;
5  import cap6.labirinto.Risultato;
```

```
6   public class RunGiocatore extends Thread {
7     private Giocatore giocatore;
8     RunGiocatore(Giocatore giocatore) {
9       this.giocatore = giocatore;
10    }
11    private static void elenca(String msg, Oggetto[] o) {
12      Console.scriviStringa(msg);
13      if (o.length == 0) {
14        Console.scriviStringa("Niente.");
15        return;
16      }
17      for (int i = 0; i < o.length; i++)
18        Console.scriviStringa(o[i].nome());
19    }
20    private static Oggetto cerca(Oggetto[] o,
21                                 String nome) {
22      for (int i = 0; i < o.length; i++) {
23        if (nome.equals(o[i].nome()))
24          return o[i];
25      }
26      return null;
27    }
28    public void run() {
29      boolean ancora = true;
30      while (ancora) {
31        Stanza stanza = (Stanza) giocatore.luogo();
32        Console.scriviStringa("**************************");
33        Console.scriviStringa(stanza.descrivi());
34        Stanza.Direzioni[] dir = stanza.direzioni();
35        if (dir.length > 0) {
36          Console.scriviStr("Puoi andare a: ");
37          for (int i = 0; i < dir.length; i++)
38            Console.scriviStr(dir[i].toString() + " ");
39          Console.nuovaLinea();
40        }
41        Oggetto[] oggetti = stanza.elenco();
42        elenca("Vedi:", oggetti);
43        Console.scriviStringa("Cosa vuoi fare?");
44        String in = Console.leggiStringa();
45        Risultato r = null;
46        if (in.equals("fine"))
```

```
47          ancora = false;
48        else if (in.equals("vai")) {
49          in = Console.leggiStringa();
50          try {
51            r = stanza.vai(
52                  giocatore,
53                  Stanza.Direzioni.valueOf(in.toUpperCase()));
54          } catch (IllegalArgumentException e) {
55            r = new Risultato(
56              false, "Non conosco la direzione " + in);
57          }
58        } else if (in.equals("prendi")) {
59          in = Console.leggiStringa();
60          Oggetto o = cerca(oggetti, in);
61          if (o == null)
62            r = new Risultato(
63              false, "Non vedo " + in + " qui.");
64          else if (o instanceof Giocatore) {
65            if (o == giocatore)
66              r = new Risultato(
67                false,
68                "Sei, per caso, il Barone di Munchausen?");
69            else
70              r = new Risultato(
71                false,
72                "E' molto abile, non riesci a prenderlo");
73          } else if (!(o instanceof Mobile))
74            r = new Risultato(false, "Non si muove.");
75          else
76            r = ((Mobile) o).sposta(stanza, giocatore);
77        } else if (in.equals("lascia")) {
78          in = Console.leggiStringa();
79          Oggetto o = (Oggetto) cerca(giocatore.elenco(), in);
80          if (o == null)
81            r = new Risultato(false, "Non possiedi " + in);
82          else
83            r = ((Mobile) o).sposta(giocatore, stanza);
84        } else if (in.equals("inventario"))
85          elenca("Possiedi: ", giocatore.elenco());
86        else
87          r = new Risultato(false, "Non capisco.");
```

```
88          if (r != null)
89            Console.scriviStringa(r.getMessage());
90        }
91      }
92    }
```

RunLadro.java:

```
1   package cap14.ladro;
2
3   import IngressoUscita.Console;
4   import cap9.prenderelasciare.*;
5   public class RunLadro extends Thread {
6     private Giocatore ladro;
7     private Stanza rifugio;
8     RunLadro(Giocatore ladro, Stanza rifugio) {
9       this.ladro = ladro;
10      this.rifugio = rifugio;
11    }
12    public void run() {
13      double r;
14      boolean ancora = true;
15      while (ancora) {
16        Stanza stanza = (Stanza) ladro.luogo();
17        if (stanza == rifugio) {
18          Oggetto[] oggetti = ladro.elenco();
19          for (int i = 0; i < oggetti.length; i++)
20            ((Mobile) oggetti[i]).sposta(ladro, stanza);
21        } else {
22          Oggetto[] oggetti = stanza.elenco();
23          for (int i = 0; i < oggetti.length; i++) {
24            if (
25              oggetti[i] instanceof Mobile &&
26              !(oggetti[i] instanceof Giocatore)) {
27              r = Math.random();
28              if (r <= 0.7)
29                ((Mobile) oggetti[i]).sposta(stanza, ladro);
30            }
31          }
32        }
33        Stanza.Direzioni[] dir = stanza.direzioni();
34        r = Math.random();
```

```
35        if (dir.length > 0) {
36          double step = 1.0 / dir.length;
37          double s = 1 - step;
38          int i = 0;
39          while (r <= s) {
40            s -= step;
41            i++;
42          }
43          stanza.vai(ladro, dir[i]);
44        }
45        if (isInterrupted())
46          ancora = false;
47        else {
48          try {
49            Thread.sleep(3000);
50          } catch (InterruptedException e) {
51            ancora = false;
52          }
53        }
54      }
55    }
56  }
```

Test.java:

```
1   package cap14.ladro;
2
3   import IngressoUscita.Console;
4   import cap9.portechiavi.*;
5   import cap9.prenderelasciare.*;
6   class Test {
7     public static void main(String[] args) {
8       Stanza s1 = new StanzaSyncr("L'ingresso");
9       Stanza s2 = new StanzaSyncr("Il corridoio");
10      Stanza s3 = new StanzaSyncr("La tana del ladro");
11      Stanza s4 = new StanzaSyncr("La stanza del tesoro");
12      Porta p1 = new Porta(s1, s2);
13      s1.collega(Stanza.Direzioni.EST, p1);
14      s2.collega(Stanza.Direzioni.OVEST, p1);
15      Porta p2 = new Porta(s2, s3);
16      s2.collega(Stanza.Direzioni.EST, p2);
17      s3.collega(Stanza.Direzioni.OVEST, p2);
```

```
18        Oggetto c1 = new MobileImplSyncr("gioielli", s4);
19        Oggetto c2 = new MobileImplSyncr("chiave", s1);
20        Porta p3 = new PortaChiusa(s2, s4, c2);
21        s2.collega(Stanza.Direzioni.SUD, p3);
22        s4.collega(Stanza.Direzioni.NORD, p3);
23        RunGiocatore r1 = new RunGiocatore(new
24                                           Giocatore("Giocatore", s1));
25        RunLadro r2 = new RunLadro(new Giocatore("Ladro", s3),
26                                   s3);
27        r1.start();
28        r2.start();
29        try {
30          r1.join();
31        } catch (InterruptedException e) {
32        }
33        r2.interrupt();
34      }
35    }
```

Note

Notare che, pur avendo usato versioni sincronizzate dei metodi di *Stanza* e *MobileImpl*, può ancora aversi un effetto indesiderato. Supponiamo che un giocatore voglia prendere un oggetto. La lista degli oggetti contenuti nella stanza corrente è ottenuta alla linea 40, mentre l'effettiva operazione che sposta l'oggetto dalla stanza al giocatore è alla linea 78. Le due operazioni sono sincronizzate *separatamente*, e nulla vieta al ladro di prendere lo stesso oggetto mentre il thread del giocatore passa dalla linea 40 alla linea 78. È in vista di questo problema che il metodo *sposta()* dell'interfaccia *Mobile*, nell'esercizio 9.8, era stato già definito con l'apparentemente inutile primo parametro *Contenitore sorg*. In questo modo, se l'oggetto è stato spostato dal momento in cui il giocatore lo ha visto al momento in cui ha deciso di prenderlo, viene evitato un nuovo spostamento dell'oggetto, e il giocatore riceve un messaggio che gli fa capire cosa è successo ("l'oggetto *x* si è spostato").

Le linee 30–34 della classe *Test* servono a fare terminare il thread del ladro una volta che è terminata l'esecuzione del thread del giocatore. In particolare, tale porzione di codice si basa sull'uso del seguente metodo della classe *Thread*:

void join() throws InterruptedException

Il thread che invoca il metodo *join()* su un altro thread si blocca fin quando quest'ultimo non termina la propria esecuzione. Un thread bloccato su un metodo *join()* può anche essere sbloccato per effetto di una chiamata al metodo *interrupt()* (in tal caso viene generata una eccezione di tipo *InterruptedException*). Nel nostro caso,

il thread del *main()* si blocca attendendo la fine del thread *r1*, quello associato al giocatore. Quando questo avviene, il thread del *main()* si sblocca e interrompe il thread del ladro attraverso una chiamata al suo metodo *interrupt()*.

Una possibile soluzione alternativa, che prescinde dall'uso del metodo *join()*, consiste nel realizzare il thread del ladro come un daemon thread ed eliminando le linee 30–34 della classe *Test*. Infatti, in questo modo, il thread del *main()* terminerebbe la propria esecuzione subito dopo aver fatto partire i thread *r1* e *r2*, ed il thread *r1*, quello del giocatore, sarebbe l'unico user thread. *r2*, essendo un daemon thread, verrebbe automaticamente terminato alla fine dell'esecuzione di *r1*.

14.6 Trova perfetti

Un numero è perfetto se è uguale alla somma dei suoi divisori propri e dell'unità. Si scriva un programma che:

1. prende in ingresso da riga di comando due parametri: il primo specifica quanti thread devono essere usati nella ricerca di numeri perfetti, il secondo quanti numeri deve esaminare ogni thread.

2. lancia i thread in modo tale che esaminino intervalli contigui (per esempio *java LanciaPerfetti 3 7000* lancia tre thread: il primo esamina gli interi che vanno da 1 a 7000, il secondo quelli che vanno da 7001 a 14000 e così via);

3. attende che tutti abbiano finito, quindi stampa l'elenco dei numeri perfetti trovati.

Suggerimenti

Può essere utile definire una classe (oltre a quella del programma principale e a quella che trova i numeri perfetti) atta a

- contenere i numeri perfetti trovati dai singoli thread;

- sincronizzare il thread del programma principale con gli altri thread.

Soluzione

Perfetti.java:

```
1  package cap14.perfetti;
2
3  class Risultato {
```

```
4     private int ancora;
5     private Elem testa;
6
7     Risultato(int n) {
8       ancora = n;
9     }
10
11    public synchronized void aggiungi(int n) {
12      System.out.println("Il thread " +
13                         Thread.currentThread() +
14                         " ha trovato il numero perfetto " + n);
15      Elem nuovo = new Elem(n);
16      if ((testa == null) || (testa.getInf() >= n)) {
17        nuovo.setNext(testa);
18        testa = nuovo;
19        return;
20      }
21      Elem temp1 = testa;
22      Elem temp2 = testa.getNext();
23      while ((temp2 != null) && (temp2.getInf() < n)) {
24        temp1 = temp1.getNext();
25        temp2 = temp2.getNext();
26      }
27      temp1.setNext(nuovo);
28      nuovo.setNext(temp2);
29    }
30
31    public synchronized void finito() {
32      System.out.println("Il thread " +
33                         Thread.currentThread() +
34                         " ha finito");
35      ancora--;
36      if (ancora == 0) {
37        notify();
38      }
39    }
40
41    public synchronized void aspetta() {
42      while (ancora > 0) {
43        try {
44          System.out.println("Il thread " +
```

```
45                              Thread.currentThread() +
46                              " aspetta.");
47          wait();
48        } catch (InterruptedException ie) {
49          ie.printStackTrace();
50        }
51      }
52    }
53
54    public String toString() {
55      String s = "(";
56      if (testa != null) {
57        s += testa;
58        for (Elem elem = testa.getNext(); elem != null;
59             elem = elem.getNext())
60          s += (", " + elem);
61      }
62      s += ")";
63      return s;
64    }
65
66    private static class Elem {
67      private int inf;
68      private Elem next;
69
70      Elem(int i) {
71        inf = i;
72      }
73
74      int getInf() {
75        return inf;
76      }
77
78      Elem getNext() {
79        return next;
80      }
81
82      void setNext(Elem e) {
83        next = e;
84      }
85
```

```
86        public String toString() {
87          return String.valueOf(inf);
88        }
89      }
90    }
91
92
93    class TrovaPerfetti extends Thread {
94      private int inf;
95      private int sup;
96      private Risultato ris;
97
98      TrovaPerfetti(Risultato r, int inf, int sup) {
99        this.inf = inf;
100       this.sup = sup;
101       ris = r;
102       System.out.println("Io cerco i perfetti tra " + inf +
103                          " e "
104                          + (sup - 1));
105     }
106
107     boolean perfetto(int x) {
108       int somma = 1;
109       for (int i = 2; i < x; i++)
110         if ((x % i) == 0) {
111           somma += i;
112         }
113       if ((x > 1) && (somma == x)) {
114         return true;
115       }
116       return false;
117     }
118
119     public void run() {
120       for (int i = inf; i < sup; i++) {
121         if (perfetto(i)) {
122           ris.aggiungi(i);
123         }
124       }
125       ris.finito();
126     }
```

```
127   }
128
129
130   class LanciaPerfetti {
131     public static void main(String[] args) {
132       if ((args == null) || (args.length != 2)) {
133         System.out.println("Uso: java LanciaPerfetti " +
134                            "numeroThread dimensioneIntervallo");
135         return;
136       }
137       int quanti = Integer.parseInt(args[0]);
138       int interv = Integer.parseInt(args[1]);
139       Risultato r = new Risultato(quanti);
140       int n = 1;
141       for (int i = 0; i < quanti; i++) {
142         TrovaPerfetti t = new TrovaPerfetti(r, n, n + interv);
143         n += interv;
144         t.start();
145       }
146       r.aspetta();
147       System.out.println("Perfetti: " + r.toString());
148     }
149   }
```

Note

La classe *Risultato* viene usata per memorizzare i numeri perfetti trovati da tutti i thread (i numeri perfetti vengono inseriti mediante il metodo *aggiungi()* in una lista i cui elementi sono istanze della classe *Elem*) e per far attendere al thread del *main()* la fine degli altri thread. La variabile membro *ancora* viene usata per tenere traccia di quanti thread devono ancora terminare il proprio compito. Il metodo *aspetta()* viene invocato dal thread del *main()* che si blocca tramite l'invocazione a *wait()* quando non tutti i thread hanno completato il loro compito. Il metodo *finito()* viene invocato quando un thread ha esaminato tutti gli interi che gli sono stati assegnati. Oltre a decrementare il valore di *ancora*, il metodo *finito()* controlla se è necessario far ripartire il thread bloccato sul metodo *aspetta()*.

14.7 Strada e rane

Un tratto stradale è caratterizzato da una larghezza e da una lunghezza. Le rane tentano di attraversare la strada nel senso della larghezza mentre le auto percorrono

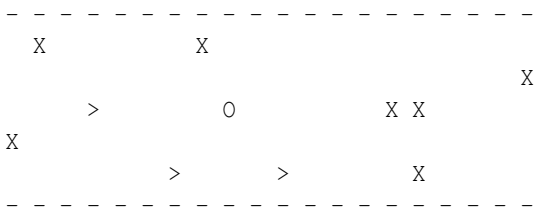


Figura 14.1: Aspetto del tratto stradale

la strada, in una sola direzione, nel senso della lunghezza. È possibile rappresentare una strada come una tabella in cui alcuni degli elementi sono occupati da rane, da auto, o da entrambi. Una possibile rappresentazione grafica della strada è mostrata in Fig 14.1.

Le auto vengono rappresentate dal carattere >. Quando una nuova auto viene aggiunta alla strada, il carattere > compare sull' estrema sinistra della rappresentazione grafica. Le auto si spostano verso destra e una volta arrivate in fondo spariscono dalla porzione di tratto stradale rappresentato. Le rane sono rappresentate dal carattere `X`. Quando una nuova rana viene aggiunta alla strada, il carattere `X` compare sulla riga più in basso (esclusa la riga che delimita la strada) della rappresentazione grafica. Le rane si muovono verso l'alto e se raggiungono l'altro lato della strada spariscono dalla rappresentazione. Quando un auto si sposta su una casella occupata da una rana, la rana viene schiacciata. Una rana schiacciata è rappresentata dal carattere `O`. Le rane schiacciate non si muovono più.

Definire la classe *Strada* in modo che sia possibile eseguire le operazioni elencate di seguito.

- *Strada(int larg, int lung)*: costruttore che crea una strada dalle dimensioni specificate. Inizialmente la strada non contiene rane né auto.
- *void aggiungiRana(int p) throws ParametroErratoException*: aggiunge una rana alla strada. Il parametro *p* indica il punto di partenza della rana (che deve essere minore della lunghezza della strada, altrimenti viene generata un'eccezione).
- *void aggiungiAuto(int q) throws ParametroErratoException*: aggiunge una auto alla strada. Il parametro *q* indica il punto di partenza dell'auto (che deve essere minore della larghezza della strada altrimenti viene generata una eccezione).
- *void aggiornaPosizioneRane()*: muove verso l'alto di una posizione ogni rana viva della strada. Se una rana viva si sposta su una posizione occupata da una rana schiacciata, la rana schiacciata sparisce dalla rappresentazione.

- *void aggiornaPosizioneAuto()*: muove verso destra di una posizione ogni auto della strada. Se l'auto si sposta su una posizione occupata da una rana viva, quest'ultima viene schiacciata. Se l'auto si sposta su una posizione occupata da una rana schiacciata, la rana schiacciata sparisce dalla rappresentazione.

- *void stampaStrada()*: stampa la strada nel modo sopra indicato.

Creare quindi 5 thread che invocano ciclicamente i metodi di un oggetto *Strada*. A tale proposito può essere utile definire 5 classi ausiliarie. Una di queste, per esempio, potrebbe essere la classe *AggiungiRane*, dotata dei metodi elencati di seguito.

- *AggiungiRane(Strada s, long x)*: crea un oggetto di tipo *AggiungiRane* che agisce sulla strada *s* ad intervalli di *x* millisecondi.

- *void run()*: esegue ciclicamente le seguenti azioni:

  - dorme per un tempo pari a *x* millisecondi;
  - genera un numero casuale intero che indica la posizione della rana da aggiungere;
  - aggiunge una rana alla strada *s*.

La strada è una risorsa condivisa da più thread, è quindi necessario assicurarsi che il suo stato rimanga sempre consistente.

Soluzione

ParametroErratoException.java:

```
1  package cap14.strada;
2
3  public class ParametroErratoException extends
4    Exception {
5    ParametroErratoException() {}
6    ParametroErratoException(String msg) {
7      super(msg);
8    }
9  }
```

Strada.java:

```
1  package cap14.strada;
2
3  import IngressoUscita.*;
4  public class Strada {
5    private final int lung;
6    private final int larg;
7    private char[][] posAuto;
8    private char[][] posRane;
9    public Strada(int lung, int larg) {
10     this.lung = lung;
11     this.larg = larg;
12     posAuto = new char[lung][larg];
13     posRane = new char[lung][larg];
14     for (int i = 0; i < lung; i++)
15       for (int j = 0; j < larg; j++) {
16         posAuto[i][j] = ' ';
17         posRane[i][j] = ' ';
18       }
19   }
20   public int getLarg() {
21     return lung;
22   }
23   public int getLung() {
24     return larg;
25   }
26   public synchronized void stampaStrada() {
27     for (int i = 0; i < lung; i++)
28       Console.scriviCar('-');
29     Console.nuovaLinea();
30     for (int i = 0; i < larg; i++) {
31       for (int j = 0; j < lung; j++)
32         if (posAuto[j][i] == ' ') {
33           Console.scriviCar(posRane[j][i]);
34         } else {
35           Console.scriviCar(posAuto[j][i]);
36         }
37       Console.nuovaLinea();
38     }
39     for (int i = 0; i < lung; i++)
40       Console.scriviCar('-');
41     Console.nuovaLinea();
```

```
42    }
43    public synchronized void aggiungiRana(int p) throws
44      ParametroErratoException {
45      if ((p < 0) || (p >= lung)) {
46        throw new ParametroErratoException();
47      }
48      if (posRane[p][larg - 1] != ' ') {
49        return;
50      }
51      posRane[p][larg - 1] = 'X';
52    }
53    public synchronized void aggiungiAuto(int p) throws
54      ParametroErratoException {
55      if ((p < 0) || (p >= larg)) {
56        throw new ParametroErratoException();
57      }
58      if (posAuto[0][p] != ' ') {
59        return;
60      }
61      posAuto[0][p] = '>';
62      if (posRane[0][p] == 'O') {
63        posRane[0][p] = ' ';
64      }
65      if (posRane[0][p] == 'X') {
66        posRane[0][p] = 'O';
67      }
68    }
69    public synchronized void aggiornaPosizioneRane() {
70      for (int i = 0; i < lung; i++)
71        if (posRane[i][0] == 'X') {
72          posRane[i][0] = ' ';
73        }
74      for (int j = 0; j < (larg - 1); j++)
75        for (int i = 0; i < lung; i++) {
76          if (posRane[i][j + 1] == 'X') {
77            posRane[i][j] = 'X';
78            posRane[i][j + 1] = ' ';
79          }
80        }
81    }
82
```

```
83     public synchronized void aggiornaPosizioneAuto() {
84       for (int j = 0; j < larg; j++)
85         if (posAuto[lung - 1][j] == '>') {
86           posAuto[lung - 1][j] = ' ';
87         }
88       for (int i = lung - 1; i > 0; i--)
89         for (int j = 0; j < larg; j++) {
90           if (posAuto[i - 1][j] == '>') {
91             posAuto[i][j] = '>';
92             posAuto[i - 1][j] = ' ';
93             if (posRane[i][j] == 'O') {
94               posRane[i][j] = ' ';
95             }
96             if (posRane[i][j] == 'X') {
97               posRane[i][j] = 'O';
98             }
99           }
100        }
101    }
102
103    public static void main(String[] args) {
104      Strada strada = new Strada(20, 8);
105      AggiungiRane ar = new AggiungiRane(strada, 500);
106      AggiungiAuto aa = new AggiungiAuto(strada, 500);
107      AggiornaRane agr = new AggiornaRane(strada, 1000);
108      AggiornaAuto aga = new AggiornaAuto(strada, 300);
109      Animatore anim = new Animatore(strada, 100);
110      ar.start();
111      aa.start();
112      agr.start();
113      aga.start();
114      anim.start();
115    }
116  }
117
118
119  abstract class ThreadCiclico extends Thread {
120    long intervallo;
121    Strada strada;
122
123    ThreadCiclico(Strada s, long interv) {
```

```
124       intervallo = interv;
125       strada = s;
126     }
127
128     abstract void azione();
129
130     public void run() {
131       try {
132         while (!isInterrupted()) {
133           sleep(intervallo);
134           azione();
135         }
136       } catch (InterruptedException ie) {
137         ie.printStackTrace();
138       }
139     }
140   }
141
142
143   class AggiungiRane extends ThreadCiclico {
144     AggiungiRane(Strada s, long interv) {
145       super(s, interv);
146     }
147
148     void azione() {
149       double d = Math.random();
150       int p = (int) (d * strada.getLarg());
151       try {
152         strada.aggiungiRana(p);
153       } catch (ParametroErratoException e) {
154         e.printStackTrace();
155       }
156     }
157   }
158
159
160   class AggiungiAuto extends ThreadCiclico {
161     AggiungiAuto(Strada s, long interv) {
162       super(s, interv);
163     }
164
```

```
165     void azione() {
166       double d = Math.random();
167       int p = (int) (d * strada.getLung());
168       try {
169         strada.aggiungiAuto(p);
170       } catch (ParametroErratoException e) {
171         e.printStackTrace();
172       }
173     }
174   }
175
176
177   class AggiornaAuto extends ThreadCiclico {
178     AggiornaAuto(Strada s, long interv) {
179       super(s, interv);
180     }
181
182     void azione() {
183       strada.aggiornaPosizioneAuto();
184     }
185   }
186
187
188   class AggiornaRane extends ThreadCiclico {
189     AggiornaRane(Strada s, long interv) {
190       super(s, interv);
191     }
192
193     void azione() {
194       strada.aggiornaPosizioneRane();
195     }
196   }
197
198
199   class Animatore extends ThreadCiclico {
200     Animatore(Strada s, long interv) {
201       super(s, interv);
202     }
203
204     void azione() {
205       strada.stampaStrada();
```

```
206       }
207   }
```

14.8 Filosofi pranzanti

Intorno ad un tavolo circolare siedono *n* filosofi ed in mezzo ad ogni coppia di filosofi c'è una sola bacchetta. I filosofi non fanno altro che pensare e mangiare. Per poter mangiare un filosofo deve riuscire a prendere entrambe le bacchette (un filosofo può prendere solo le bacchette che gli sono vicine), ma due filosofi non possono prendere contemporaneamente la stessa bacchetta. Realizzare un programma in cui ogni filosofo (thread) periodicamente esegue le seguenti azioni: *i*) pensa per un certo tempo; *ii*) prende la prima bacchetta; *iii*) aspetta un po' di tempo, quindi prende la seconda bacchetta; *iv*) mangia per un po' di tempo, *v*) rilascia le bacchette.

Il programma deve chiedere da tastiera il numero di filosofi che devono essere simulati. Ogni volta che un filosofo prende o lascia una bacchetta, il programma deve stampare su video una linea che rappresenta lo stato del sistema, come nel seguente esempio:

```
|  0  | |  1 X| |  2  | |  3  | |  4  |
|X 0  | |  1 X| |  2  | |  3  | |  4  |
|X 0  | |  1 X| |  2  | |X 3  | |  4  |
|X 0  | |  1 X| |  2  | |X 3  | |X 4  |
|X 0  | |X 1 X| |  2  | |X 3  | |X 4  |
|X 0  | |  1 X| |  2  | |X 3  | |X 4  |
|X 0  | |  1  | |  2  | |X 3  | |X 4  |
|X 0  | |  1  | |X 2  | |X 3  | |X 4  |
|X 0 X| |  1  | |X 2  | |X 3  | |X 4  |
```

Ogni linea rappresenta lo stato del sistema ad un certo istante. Ogni filosofo è rappresentato da una stringa "`|` *n* `|`", dove *n* è numero che identifica il filosofo, e la stringa contiene un carattere "`X`" a destra o a sinistra, a seconda che il filosofo *n* abbia preso o meno la bacchetta destra o sinistra.

Soluzione

Bacchetta.java:

```
1   package cap14.filosofi;
2
3   public class Bacchetta {
4     protected boolean inUso;
```

```
5
6     public synchronized void prendi() throws
7       InterruptedException {
8       while (inUso)
9         wait();
10      inUso = true;
11    }
12
13    public synchronized void rilascia() {
14      inUso = false;
15      notify();
16    }
17  }
```

Filosofo.java:

```
1   package cap14.filosofi;
2
3   public class Filosofo extends Thread {
4     protected static long MAX_PENSA = 100L;
5     protected static long MAX_BAC = 100L;
6     protected static long MAX_MANGIA = 100L;
7     protected Bacchetta sinistra;
8     protected Bacchetta destra;
9     protected boolean hoSin;
10    protected boolean hoDes;
11    protected int ident;
12    protected TavolaFilosofi tav;
13
14    public Filosofo(TavolaFilosofi t, int id) {
15      tav = t;
16      sinistra = tav.rifBacchettaSin(id);
17      destra = tav.rifBacchettaDes(id);
18      ident = id;
19      hoSin = false;
20      hoDes = false;
21      tav.setStato(ident, stato());
22    }
23
24    protected long tempoPensa() {
25      return (long) (Math.random() * MAX_PENSA) + 1;
26    }
```

```
27
28      protected long tempoMangia() {
29        return (long) (Math.random() * MAX_MANGIA) + 1;
30      }
31
32      protected long tempoBac() {
33        return (long) (Math.random() * MAX_BAC) + 1;
34      }
35
36      protected void prendi(Bacchetta b) throws
37        InterruptedException {
38        b.prendi();
39        if (b == destra) {
40          hoDes = true;
41        } else {
42          hoSin = true;
43        }
44        tav.setStato(ident, stato());
45        tav.stampa();
46      }
47
48      protected void lascia(Bacchetta b) throws
49        InterruptedException {
50        b.rilascia();
51        if (b == destra) {
52          hoDes = false;
53        } else {
54          hoSin = false;
55        }
56        tav.setStato(ident, stato());
57        tav.stampa();
58      }
59
60      protected void prendiBacchette() throws
61        InterruptedException {
62        if (Math.random() < 0.5) {
63          prendi(sinistra);
64          Thread.sleep(tempoBac());
65          prendi(destra);
66        } else {
67          prendi(destra);
```

```
68          Thread.sleep(tempoBac());
69          prendi(sinistra);
70        }
71      }
72
73      protected final void mangia() throws
74        InterruptedException {
75        Thread.sleep(tempoMangia());
76      }
77
78      protected void lasciaBacchette() throws
79        InterruptedException {
80        lascia(sinistra);
81        lascia(destra);
82      }
83
84      protected final void pensa() throws
85        InterruptedException {
86        Thread.sleep(tempoPensa());
87        tav.setStato(ident, stato());
88      }
89
90      public String stato() {
91        String s = (hoSin ? "|X " : "|  ") + ident +
92                   (hoDes ? " X|" : "  |");
93        return s;
94      }
95
96      public final void run() {
97        while (true) {
98          try {
99            pensa();
100           prendiBacchette();
101           mangia();
102           lasciaBacchette();
103         } catch (InterruptedException e) {
104           e.printStackTrace();
105         }
106       }
107     }
108   }
```

TavolaFilosofi.java:

```
1  package cap14.filosofi;
2
3  import IngressoUscita.Console;
4  public class TavolaFilosofi {
5    protected int num;
6    protected Bacchetta[] bac;
7    protected Filosofo[] fil;
8    public String[] rap;
9    protected TavolaFilosofi() {
10   }
11   public TavolaFilosofi(int n) {
12     num = n;
13     bac = new Bacchetta[num];
14     fil = new Filosofo[num];
15     rap = new String[num];
16     for (int i = 0; i < n; i++)
17       bac[i] = new Bacchetta();
18     for (int i = 0; i < n; i++) {
19       fil[i] = new Filosofo(this, i);
20       fil[i].start();
21     }
22   }
23   public Bacchetta rifBacchettaSin(int i) {
24     return bac[i];
25   }
26   public Bacchetta rifBacchettaDes(int i) {
27     return bac[(i + 1) % num];
28   }
29   public synchronized void setStato(int i, String s) {
30     rap[i] = s;
31   }
32   public synchronized void stampa() {
33     for (int i = 0; i < num; i++)
34       Console.scriviStr(rap[i]);
35     Console.nuovaLinea();
36   }
37 }
```

FilosofiPranzanti.java:

```
1  package cap14.filosofi;
```

```
2
3   import IngressoUscita.Console;
4   public class FilosofiPranzanti {
5     public static void main(String[] args) {
6       Console.scriviStringa("Quanti filosofi ?");
7       int n = Console.leggiIntero();
8       TavolaFilosofi tf = new TavolaFilosofi(n);
9     }
10  }
```

Note

Se proviamo ad eseguire il programma ci accorgiamo che, dopo un tempo variabile, il programma si ferma. Per esempio, con 5 filosofi, potremmo ottenere una serie di linee che termina in questo modo:

```
|X 0   | |X 1   | |X 2   | |X 3   | |X 4   |
```

In questo stato, ogni filosofo ha preso la bacchetta sinistra e sta aspettando che il filosofo alla sua destra lasci l'altra bacchetta. Poiché il tavolo è circolare, non c'è modo di uscire da questa situazione, che viene detta *deadlock*.

Ci sono vari modi di risolvere il problema, cambiando la strategia con cui ogni filosofo decide di prendere o non prendere una bacchetta. In molti di questi modi, però, può presentarsi un diverso tipo di problema, detto *starvation*. La starvation è una situazione in cui alcuni filosofi non riescono mai a mangiare, o mangiano molto meno degli altri. La soluzione dell'esercizio è stata scritta in modo che sia possibile sperimentare con diverse soluzioni, estendendo la classe *Filosofo* (o le altre) e modificando il comportamento dei metodi *prendiBacchette()* e *lasciaBacchette()*. Le varie soluzioni, però, non devono modificare ciò che avviene mentre un filosofo mangia o pensa, ed è per questo che i relativi metodi sono stati definiti *final*.

14.9 Filosofi pranzanti: deadlock e starvation

Si estandano le classi definite nella soluzione dell'esercizio 14.8, in modo che non si verifichino problemi di deadlock e starvation.

Il programma deve stampare una linea che rappresenti lo stato del sistema, che mostri anche il numero di volte che ogni filosofo ha mangiato (si assuma che il filosofo mangerà sicuramente, ogni volta che riesce a prendere entrambe le bacchette).

Suggerimenti

Una possibile soluzione è quella di fare in modo che una delle bacchette sia "marcata". Se un filosofo prende per prima una bacchetta marcata deve lasciarla e provare a prendere l'altra (se, invece, la prende per seconda, prosegue normalmente e comincia a mangiare). Quando il filosofo che ha preso la bacchetta marcata ha finito di mangiare, posa entrambe le bacchette scambiando la destra con la sinistra.

Soluzione

BacchettaMarcata.java:

```
1  package cap14.filosofiND;
2
3  import cap14.filosofi.Bacchetta;
4  public class BacchettaMarcata extends Bacchetta {
5    protected boolean marcata;
6    public BacchettaMarcata(boolean m) {
7      marcata = m;
8    }
9    public boolean isMarcata() {
10     return marcata;
11   }
12   public void marca() {
13     marcata = true;
14   }
15   public void smarca() {
16     marcata = false;
17   }
18 }
```

FilosofoND.java:

```
1  package cap14.filosofiND;
2
3  import cap14.filosofi.*;
4  public class FilosofoND extends Filosofo {
5    int nMangiate;
6    public FilosofoND(TavolaFilosofiND t, int id) {
7      super(t, id);
8      nMangiate = 0;
9    }
10   protected void prendiBacchette() throws
```

```
11        InterruptedException {
12        boolean pSin = Math.random() < 0.5;
13        for (;;) {
14          if (pSin) {
15            prendi(sinistra);
16            if (((BacchettaMarcata) sinistra).isMarcata()) {
17              lascia(sinistra);
18              pSin = false;
19            } else {
20              break;
21            }
22          } else {
23            prendi(destra);
24            if (((BacchettaMarcata) destra).isMarcata()) {
25              lascia(destra);
26              pSin = true;
27            } else {
28              break;
29            }
30          }
31          Thread.sleep(tempoBac());
32        }
33        Thread.sleep(tempoBac());
34        if (hoSin) {
35          prendi(destra);
36        } else {
37          prendi(sinistra);
38        }
39        nMangiate++;
40      }
41      protected void lasciaBacchette() throws
42        InterruptedException {
43        BacchettaMarcata sin = (BacchettaMarcata) sinistra;
44        BacchettaMarcata des = (BacchettaMarcata) destra;
45        if (sin.isMarcata()) {
46          des.marca();
47          sin.smarca();
48        } else if (des.isMarcata()) {
49          des.smarca();
50          sin.marca();
51        }
```

```
52      super.lasciaBacchette();
53    }
54    public String stato() {
55      String sSin = "|  ";
56      String sDes = "  |";
57      if (hoSin) {
58        sSin = (((BacchettaMarcata) sinistra).isMarcata() ?
59                "|O " :
60                "|X ");
61      }
62      if (hoDes) {
63        sDes = (((BacchettaMarcata) destra).isMarcata() ?
64                " O|" :
65                " X|");
66      }
67      return sSin + nMangiate + sDes;
68    }
69  }
```

TavolaFilosofiND.java:

```
1   package cap14.filosofiND;
2
3   import cap14.filosofi.*;
4   public class TavolaFilosofiND extends TavolaFilosofi {
5     public TavolaFilosofiND(int n) {
6       num = n;
7       bac = new Bacchetta[num];
8       fil = new Filosofo[num];
9       rap = new String[num];
10      for (int i = 0; i < n; i++)
11        bac[i] = new BacchettaMarcata(i == 0);
12      for (int i = 0; i < n; i++) {
13        fil[i] = new FilosofoND(this, i);
14        fil[i].start();
15      }
16    }
17  }
```

FilosofiPranzantiND.java:

```
1   package cap14.filosofiND;
```

```
2
3   import IngressoUscita.Console;
4   public class FilosofiPranzantiND {
5     public static void main(String[] args) {
6       Console.scriviStringa("Quanti filosofi ?");
7       int n = Console.leggiIntero();
8       TavolaFilosofiND tf = new TavolaFilosofiND(n);
9     }
10  }
```

Note

La soluzione adottata prevede che una bacchetta sia marcata. Nella ridefinizione del metodo *prendiBacchette()*, si sceglie casualmente se prendere la destra o la sinistra (linea 13). Supponendo di aver scelto la sinistra, si prova a prenderla (linea 16) e, se è marcata, la si rilascia e si riprova con la destra (linee 17–20). Notare che bisogna ricontrollare che anche la destra non sia marcata, in quanto dal momento in cui il filosofo posa la bacchetta marcata alla sua sinistra, al momento in cui prova a prendere la bacchetta alla sua destra, la bacchetta marcata potrebbe fare il giro di tutto il tavolo e arrivare alla sua destra. Per questo motivo, il metodo *prendiBacchette()* è organizzato come un ciclo infinito (linee 14–32), da cui si esce solo quando il filosofo riesce a prendere una bacchetta non marcata.

Provando a lanciare questo programma, otteniamo un output del tipo (per 5 filosofi):

```
                    ...
|  88  | |  89 X| |  87  | |X 84  | |X 84  |
|  88  | |  89 X| |  87  | |X 84  | |X 84 O|
|  88  | |X 89 X| |  87  | |X 84  | |X 84 O|
|  88  | |X 89 X| |  87  | |X 84  | |  85 X|
|  88  | |X 89 X| |  87  | |X 84  | |  85  |
|  88  | |X 89 X| |  87  | |X 84 O| |  85  |
|X 88  | |X 89 X| |  87  | |X 84 O| |  85  |
|X 88  | |X 89 X| |  87  | |  85 X| |  85  |
|X 88  | |X 89 X| |  87  | |  85  | |  85  |
|X 88  | |X 89 X| |  87 O| |  85  | |  85  |
|X 88  | |X 89 X| |  87  | |  85  | |  85  |
|X 88  | |X 89 X| |  87  | |  85  | |X 85  |
```

dove i numeri rappresentano il numero di volte che un filosofo ha mangiato, e il simbolo "O" rappresenta la bacchetta marcata.

15. Programmazione di rete

15.1 Calcolatore

Si realizzi un programma servitore che offre i servizi di somma, sottrazione, moltiplicazione e divisione di due numeri interi. Si realizzi inoltre un programma cliente che usa i servizi esportati. La comunicazione tra i due programmi avviene attraverso socket TCP.

Suggerimenti

La divisione di un numero per zero solleva una *ArithmeticException*, trattare opportunamente tale situazione.

Soluzione

Op.java:

```
1  package cap15.calcolatore;
2
3  public enum Op {
4    USCITA(0), SOMMA(1), SOTTRAZIONE(2), DIVISIONE(3),
5    MOLTIPLICAZIONE(4), OK(5), ERROR(6);
6    private final int value;
7    private Op(int v) {
8      value = v;
9    }
10   public int toInt() {
11     return value;
12   }
```

```
13    public static Op fromValue(int b) {
14      for (Op o : Op.values())
15        if (o.value == b)
16          return o;
17      return null;
18    }
19  }
```

CalcServitore.java:

```
1   package cap15.calcolatore;
2
3   import java.io.*;
4   import java.net.*;
5   public class CalcServitore {
6     private final int port;
7     private DataInputStream dis;
8     private DataOutputStream dos;
9     CalcServitore(int p) {
10      port = p;
11    }
12    CalcServitore() {
13      this(10000);
14    }
15    public void startService() {
16      ServerSocket serv = null;
17      try {
18        serv = new ServerSocket(port);
19      } catch (IOException ioe) {
20        System.out.println(
21          "Non riesco a mettermi in ascolto sulla porta specificata");
22        System.exit(1);
23      }
24      while (true) {
25        try {
26          Socket s = serv.accept();
27          System.out.println("Si e' connesso un cliente");
28          dis = new DataInputStream(s.getInputStream());
29          dos = new DataOutputStream(s.getOutputStream());
30          Op op;
31          boolean continua = true;
32          while (continua) {
```

```
33              op = Op.fromValue(dis.readInt());
34              switch (op) {
35              case SOMMA:
36                somma();
37                break;
38              case SOTTRAZIONE:
39                sottrazione();
40                break;
41              case DIVISIONE:
42                divisione();
43                break;
44              case MOLTIPLICAZIONE:
45                moltiplicazione();
46                break;
47              case USCITA:
48                continua = false;
49                break;
50              }
51            }
52          } catch (IOException ioe) {
53            System.out.println("Problemi durante la comunicazione: "
54                               +
55                               ioe.getMessage());
56          } finally {
57            try {
58              if (dis != null) {
59                dis.close();
60              }
61              if (dos != null) {
62                dos.close();
63              }
64            } catch (IOException ioe) {
65            }
66          }
67        }
68      }
69      private void somma() throws IOException {
70        int a = dis.readInt();
71        int b = dis.readInt();
72        int r = a + b;
73        dos.writeInt(r);
```

```
74      }
75      private void sottrazione() throws IOException {
76        int a = dis.readInt();
77        int b = dis.readInt();
78        int r = a - b;
79        dos.writeInt(r);
80      }
81      private void divisione() throws IOException {
82        int a = dis.readInt();
83        int b = dis.readInt();
84        int r;
85        try {
86          r = a / b;
87        } catch (ArithmeticException e) {
88          dos.writeInt(Op.ERROR.toInt());
89          return;
90        }
91        dos.writeInt(Op.OK.toInt());
92        dos.writeInt(r);
93      }
94      private void moltiplicazione() throws IOException {
95        int a = dis.readInt();
96        int b = dis.readInt();
97        int r = a * b;
98        dos.writeInt(r);
99      }
100
101     public static void main(String[] args) {
102       CalcServitore s;
103       if (args.length == 0) {
104         s = new CalcServitore();
105       } else {
106         s = new CalcServitore(Integer.parseInt(args[0]));
107       }
108       s.startService();
109     }
110   }
```

CalcCliente.java:

```
1   package cap15.calcolatore;
2
```

```
3  import java.io.*;
4  import java.net.*;
5  public class CalcCliente {
6    private String serverAddress;
7    private int serverPort;
8    private Socket s;
9    private DataInputStream dis;
10   private DataOutputStream dos;
11   public CalcCliente(String h,
12                      int i) throws IOException {
13     serverAddress = h;
14     serverPort = i;
15     s = new Socket(serverAddress, serverPort);
16     dis = new DataInputStream(s.getInputStream());
17     dos = new DataOutputStream(s.getOutputStream());
18   }
19   public CalcCliente() throws IOException {
20     this("localhost", 10000);
21   }
22   public void uscita() throws IOException {
23     dos.writeInt(Op.USCITA.toInt());
24   }
25   public int sum(int a, int b) throws IOException {
26     dos.writeInt(Op.SOMMA.toInt());
27     dos.writeInt(a);
28     dos.writeInt(b);
29     int r = dis.readInt();
30     return r;
31   }
32   public int sub(int a, int b) throws IOException {
33     dos.writeInt(Op.SOTTRAZIONE.toInt());
34     dos.writeInt(a);
35     dos.writeInt(b);
36     int r = dis.readInt();
37     return r;
38   }
39   public int mul(int a, int b) throws IOException {
40     dos.writeInt(Op.MOLTIPLICAZIONE.toInt());
41     dos.writeInt(a);
42     dos.writeInt(b);
43     int r = dis.readInt();
```

```
44      return r;
45    }
46    public int div(int a, int b)
47    throws IOException, ArithmeticException {
48      dos.writeInt(Op.DIVISIONE.toInt());
49      dos.writeInt(a);
50      dos.writeInt(b);
51      Op cond = Op.fromValue(dis.readInt());
52      if (cond == Op.OK) {
53        int r = dis.readInt();
54        return r;
55      }
56      throw new ArithmeticException("Divisione per zero");
57    }
58  }
```

ProvaCliente.java:

```
1   package cap15.calcolatore;
2
3   import java.io.*;
4   public class ProvaCliente {
5     public static void main(String[] args) {
6       CalcCliente cc = null;
7       try {
8         cc = new CalcCliente();
9         System.out.println("3+5=" + cc.sum(3, 5));
10        System.out.println("6*9=" + cc.mul(6, 9));
11        System.out.println("7/0=" + cc.div(7, 0));
12      } catch (IOException e) {
13        System.out.println("Problemi di comunicazione ...");
14      } finally {
15        if (cc != null) {
16          try {
17            cc.uscita();
18          } catch (IOException ioe) {
19          }
20        }
21      }
22    }
23  }
```

15.2 Calcolatore concorrente

Riscrivere il servitore dell'esercizio precedente in modo tale che sia in grado di servire più clienti contemporaneamente. Notare che, se il protocollo di comunicazione tra il cliente ed il servitore non cambia, non è necessario riscrivere il cliente (si può usare quello dell'esercizio precedente).

Soluzione

CalcServitore2.java:

```
1  package cap15.calcconc;
2
3  import java.io.*;
4  import java.net.*;
5  public class CalcServitore2 {
6    private final int port;
7    CalcServitore2(int p) {
8      port = p;
9    }
10   CalcServitore2() {
11     this(10000);
12   }
13   public void startService() {
14     ServerSocket ss = null;
15     try {
16       ss = new ServerSocket(port);
17     } catch (IOException ioe) {
18       System.out.println(
19         "Non riesco a mettermi in ascolto sulla porta specificata");
20       System.exit(1);
21     }
22     while (true) {
23       try {
24         Socket s = ss.accept();
25         CalcFiglio calc = new CalcFiglio(s);
26         calc.start();
27       } catch (IOException ioe) {
28         System.out.println("Problemi durante la comunicazione con il cliente: "
29                            +
30                            ioe.getMessage());
31       }
```

```
32      }
33    }
34    public static void main(String[] args) {
35      CalcServitore2 s = new CalcServitore2();
36      s.startService();
37    }
38  }
```

CalcFiglio.java:

```
1   package cap15.calcconc;
2
3   import cap15.calcolatore.Op;
4   import java.io.*;
5   import java.net.*;
6   public class CalcFiglio extends Thread {
7     private Socket s;
8     private DataInputStream dis;
9     private DataOutputStream dos;
10    CalcFiglio(Socket s) {
11      this.s = s;
12    }
13    public void run() {
14      service();
15    }
16    private void service() {
17      try {
18        System.out.println("Si e' connesso un cliente");
19        dis = new DataInputStream(s.getInputStream());
20        dos = new DataOutputStream(s.getOutputStream());
21        Op op;
22        boolean continua = true;
23        while (continua) {
24          op = Op.fromValue(dis.readInt());
25          switch (op) {
26          case SOMMA:
27            somma();
28            break;
29          case SOTTRAZIONE:
30            sottrazione();
31            break;
32          case DIVISIONE:
```

```
33              divisione();
34              break;
35            case MOLTIPLICAZIONE:
36              moltiplicazione();
37              break;
38            case USCITA:
39              continua = false;
40              break;
41            }
42          }
43          System.out.println("Sessione terminata");
44          dis.close();
45          dos.close();
46        } catch (IOException ioe) {
47          System.out.println("Problemi durante la comunicazione con il cliente"
48                             +
49                             ioe.getMessage());
50        } finally {
51          try {
52            if (dis != null) {
53              dis.close();
54            }
55            if (dos != null) {
56              dos.close();
57            }
58          } catch (IOException ioe) {
59          }
60        }
61      }
62      private void somma() throws IOException {
63        int a = dis.readInt();
64        int b = dis.readInt();
65        int r = a + b;
66        dos.writeInt(r);
67      }
68      private void sottrazione() throws IOException {
69        int a = dis.readInt();
70        int b = dis.readInt();
71        int r = a - b;
72        dos.writeInt(r);
73      }
```

```
74      private void divisione() throws IOException {
75        int a = dis.readInt();
76        int b = dis.readInt();
77        int r;
78        try {
79          r = a / b;
80        } catch (ArithmeticException e) {
81          dos.writeInt(Op.ERROR.toInt());
82          return;
83        }
84        dos.writeInt(Op.OK.toInt());
85        dos.writeInt(r);
86      }
87      private void moltiplicazione() throws IOException {
88        int a = dis.readInt();
89        int b = dis.readInt();
90        int r = a * b;
91        dos.writeInt(r);
92      }
93    }
```

15.3 Chiacchiere

Realizzare un sistema distribuito per lo scambio di brevi messaggi testuali in cui la comunicazione tra il programma mittente e quello ricevente avviene attraverso pacchetti UDP. In particolare il programma ricevitore si pone in ascolto sulla porta specificata da riga di comando, e ogni volta che riceve un pacchetto ne estrae il messaggio testuale e lo visualizza sullo schermo. Il programma mittente esegue un ciclo in cui attende l'immissione da parte dell'utente di una nuova riga di testo e la invia al programma ricevitore (l'indirizzo e la porta del ricevitore vengono indicati al programma mittente attraverso due argomenti della riga di comando). Trascurare la eventuale perdita di pacchetti.

Suggerimenti

Il metodo *byte[] getBytes()* della classe *String* restituisce la rappresentazione in byte dell'oggetto stringa a cui viene applicato in accordo alla codifica dei caratteri predefinita della macchina. Il costruttore *String(byte[] b)* crea un oggetto stringa partendo dall'array di byte specificato e utilizzando la codifica dei caratteri predefinita della macchina. Si assuma, per semplicità, che la codifica dei caratteri del mittente e del ricevitore siano uguali.

Soluzione

Mittente.java:

```
1  package cap15.chiac;
2
3  import java.io.*;
4  import java.net.*;
5  public class Mittente {
6    protected static int PORTA_PREDEFINITA = 9000;
7    protected int portaDest;
8    protected String indirizzoDest;
9    protected DatagramSocket ds;
10   protected DatagramPacket dp;
11   protected byte[] buf;
12   public Mittente(String addr) {
13     this(addr, PORTA_PREDEFINITA);
14   }
15   public Mittente(String addr, int pd) {
16     indirizzoDest = addr;
17     portaDest = pd;
18   }
19   public void attiva() throws IOException {
20     ds = new DatagramSocket();
21   }
22   public void esegui() throws IOException {
23     InetAddress iaDest = InetAddress.getByName(
24                             indirizzoDest);
25     BufferedReader br = new BufferedReader(new
26                                          InputStreamReader(System.in));
27     while (true) {
28       String s = br.readLine();
29       if (s.equals(".")) {
30         break;
31       }
32       byte[] msg = s.getBytes();
33       dp = new DatagramPacket(msg, msg.length, iaDest,
34                               portaDest);
35       ds.send(dp);
36     }
37   }
38   public static void main(String[] args) {
```

```
39      if ((args.length < 1) || (args.length > 2)) {
40        System.out.println("Uso: java Mittente hostDestinatario <porta>");
41        return;
42      }
43      Mittente c;
44      if (args.length == 1) {
45        c = new Mittente(args[0]);
46      } else {
47        c = new Mittente(args[0], Integer.parseInt(args[1]));
48      }
49      System.out.println("Per terminare scrivere una riga che contiene "
50                         +
51                         "solo il carattere \'.\', quindi premere invio");
52      try {
53        c.attiva();
54        c.esegui();
55      } catch (IOException e) {
56        System.err.println("Errore: ");
57        e.printStackTrace();
58      }
59    }
60  }
```

Ricevitore.java:

```
1   package cap15.chiac;
2
3   import java.io.*;
4   import java.net.*;
5   public class Ricevitore {
6     protected static int PORTA_PREDEFINITA = 9000;
7     protected static int LUN_BUF = 65536;
8     protected int porta;
9     protected byte[] buf;
10    protected DatagramSocket ds;
11    protected DatagramPacket dp;
12    public Ricevitore() {
13      this(PORTA_PREDEFINITA);
14    }
15    public Ricevitore(int p) {
16      porta = p;
17      buf = new byte[LUN_BUF];
```

```
18        dp = new DatagramPacket(buf, buf.length);
19      }
20      public void attiva() throws IOException {
21        ds = new DatagramSocket(porta);
22      }
23      public void esegui() throws IOException {
24        while (true) {
25          ds.receive(dp);
26          byte[] data = dp.getData();
27          int l = dp.getLength();
28          String msg = new String(data, 0, l);
29          System.out.println(msg);
30        }
31      }
32      public static void main(String[] args) {
33        Ricevitore r;
34        if (args.length == 0) {
35          r = new Ricevitore();
36        } else if (args.length == 1) {
37          r = new Ricevitore(Integer.parseInt(args[0]));
38        } else {
39          System.out.println("Uso: java Ricevitore <porta>");
40          return;
41        }
42        try {
43          r.attiva();
44          r.esegui();
45        } catch (IOException ioe) {
46          System.err.println("Errore di rete: " +
47                             ioe.getMessage());
48        }
49      }
50    }
```

15.4 Chiacchiere (con tutti)

Vogliamo estendere il sistema sviluppato nell'esercizio precedente in modo tale che ogni messaggio spedito venga recapitato a tutti i ricevitori presenti in un certo "raggio", sfruttando i meccanismi messi a disposizione dal multicast.

Soluzione

RicevitoreMulti.java:

```
1  package cap15.multichiac;
2
3  import cap15.chiac.Ricevitore;
4  import java.io.*;
5  import java.net.*;
6  public class RicevitoreMulti extends Ricevitore {
7    protected String gruppo;
8    public RicevitoreMulti(String g) {
9      this(g, PORTA_PREDEFINITA);
10   }
11   public RicevitoreMulti(String g, int p) {
12     super(p);
13     gruppo = g;
14   }
15   public void attiva() throws IOException {
16     ds = new MulticastSocket(porta);
17     ((MulticastSocket) ds).joinGroup(
18       InetAddress.getByName(gruppo));
19   }
20   public static void main(String[] args) {
21     RicevitoreMulti r;
22     if (args.length == 1)
23       r = new RicevitoreMulti(args[0]);
24     else if (args.length == 2)
25       r = new RicevitoreMulti(
26         args[0], Integer.parseInt(args[1]));
27     else {
28       System.out.println(
29         "Uso: java RicevitoreMulti gruppoMulti <porta>");
30       return;
31     }
32     try {
33       r.attiva();
34       r.esegui();
35     } catch (IOException ioe) {
36       System.err.println(
37         "Errore di rete: " + ioe.getMessage());
38     }
```

```
39      }
40    }
```

MittenteMulti.java:

```
1   package cap15.multichiac;
2
3   import cap15.chiac.Mittente;
4   import java.io.*;
5   import java.net.*;
6   public class MittenteMulti extends Mittente {
7     protected int ttl;
8     public MittenteMulti(String addr, int pd, int ttl) {
9       super(addr, pd);
10      this.ttl = ttl;
11    }
12    public void attiva() throws IOException {
13      ds = new MulticastSocket();
14      ((MulticastSocket) ds).setTimeToLive(ttl);
15    }
16    public static void main(String[] args) {
17      if (args.length != 3) {
18        System.out.println("Uso: java MittenteMulti gruppoMulti portaDest ttl");
19        return;
20      }
21      MittenteMulti c = new MittenteMulti(args[0],
22                                          Integer.parseInt(args[1]),
23                                          Integer.parseInt(args[2]));
24      System.out.println("Per terminare scrivere una riga che contiene "
25                         +
26                         "solo il carattere \'.\', quindi premere invio");
27      try {
28        c.attiva();
29        c.esegui();
30      } catch (IOException e) {
31        System.err.println("Errore: " + e.getMessage());
32      }
33    }
34  }
```

15.5 Agenda remota

Vogliamo realizzare un sistema per la gestione remota di una agenda di appuntamenti. Il sistema si compone di un programma cliente e di un programma servitore. Quando viene avviato, il cliente chiede all'utente di inserire il proprio nome, quindi recupera dal servitore l'insieme degli appuntamenti relativi a tale utente. A questo punto, viene visualizzata una interfaccia testuale simile alla seguente che consente di manipolare l'elenco di appuntamenti:

```
Appuntamenti:
0) 12/3/2007, alle 12: Idraulico
1) 15/3/2007, alle 10: Esame patente
2) 26/3/2007, alle 18: Arriva zia Caterina

a) aggiungi un appuntamento
b) rimuovi un appuntamento
c) salva sul server
d) esci
```

Il servitore, all'avvio, si pone in ascolto su una porta predefinita e attende la connessione da parte di un cliente. Quando un cliente si connette, il servitore recupera da un file i dati relativi agli appuntamenti dell'utente che si è appena connesso (per semplicità i dati relativi ad un dato utente possono essere memorizzati un un file che ha il suo stesso nome, senza preoccuparci di problemi di omonimia). Il server, su richiesta del cliente, provvede a rimpiazzare i dati su disco con quelli modificati dall'utente.

Per semplicità possiamo realizzare il servitore in modo tale da poter gestire un solo cliente alla volta.

Suggerimenti

Si consiglia di realizzare le seguenti classi:

- *Data*: in grado di memorizzare la data di un appuntamento. Se la classe *Data* implementa l'interfaccia *Comparable*<*Data*>, l'ordinamento temporale tra appuntamenti può essere realizzato semplicemente. L'interfaccia *Comparable* è così fatta:

  ```
  package java.lang;

  public interface Comparable<E>{
    public abstract int compareTo(E e);
  }
  ```

dove il metodo *compareTo()* deve essere implementato in modo tale da restituire un numero negativo quando l'oggetto implicito precede quello passato come argomento, un numero positivo quando lo segue e zero quando i due oggetti sono equivalenti dal punto di vista dell'ordinamento.

- *Appuntamento*: una classe che memorizza i dati di un singolo appuntamento. Anche in questo caso può essere utile implementare l'interfaccia *Comparable* (tra istanze della classe *Appuntamento*).
- *Agenda*: una classe che raggruppa gli appuntamenti di un dato utente. Rendendo la classe *Agenda* serializzabile è possibile trasferirne istanze tra il programma cliente e quello servitore.

Soluzione

Data.java:

```
1  package cap15.app;
2
3  import java.io.*;
4  import java.util.*;
5  public class Data implements Serializable,
6    Comparable<Data> {
7    private int giorno;
8    private int mese;
9    private int anno;
10   public Data(int g, int m, int a) {
11     giorno = g;
12     mese = m;
13     anno = a;
14   }
15   public Data(String d) {
16     Scanner s = new Scanner(d);
17     s.useDelimiter("/");
18     giorno = s.nextInt();
19     mese = s.nextInt();
20     anno = s.nextInt();
21   }
22   public int compareTo(Data d) {
23     if (anno != d.anno)
24       return anno - d.anno;
25     if (mese != d.mese)
```

```
26        return mese - d.mese;
27      return giorno - d.giorno;
28    }
29    public String toString() {
30      return giorno + "/" + mese + "/" + anno;
31    }
32  }
```

Appuntamento.java:

```
1   package cap15.app;
2
3   import java.io.*;
4   public class Appuntamento implements Serializable,
5     Comparable<Appuntamento> {
6     private String titolo;
7     private Data data;
8     private int ora;
9     public Appuntamento(String t, Data d, int o) {
10      titolo = t;
11      data = d;
12      ora = o;
13    }
14    public String toString() {
15      return data + ", alle " + ora + ": " + titolo;
16    }
17    public int compareTo(Appuntamento a) {
18      int r = data.compareTo(a.data);
19      if (r == 0)
20        return ora - a.ora;
21      else
22        return r;
23    }
24  }
```

IndiceNonValidoException.java:

```
1   package cap15.app;
2
3   public class IndiceNonValidoException
4     extends RuntimeException {
5     public IndiceNonValidoException() {
```

```
6       super();
7     }
8     public IndiceNonValidoException(String s) {
9       super(s);
10    }
11  }
```

Agenda.java:

```
1   package cap15.app;
2
3   import java.io.*;
4   public class Agenda implements Serializable {
5     private static int DEF_N = 10;
6     private Appuntamento[] app = new Appuntamento[DEF_N];
7     private int n;
8     public void inserisci(Appuntamento a) {
9       if (n == app.length) {
10        cresci();
11      }
12      int i = 0;
13      for (; (i < n) && (a.compareTo(app[i]) > 0); i++)
14        ;
15      for (int j = n; j > i; j--)
16        app[j] = app[j - 1];
17      app[i] = a;
18      n++;
19    }
20    public void rimuovi(int k) {
21      if ((k < 0) || (k >= n)) {
22        throw new IndiceNonValidoException();
23      }
24      for (int i = k; i < (n - 1); i++)
25        app[i] = app[i + 1];
26      app[n - 1] = null;
27      n--;
28      if ((n < (app.length / 2)) && (app.length > DEF_N)) {
29        diminuisci();
30      }
31    }
32    private void cresci() {
33      Appuntamento[] tmp = new Appuntamento[app.length * 2];
```

```
34      for (int i = 0; i < n; i++)
35        tmp[i] = app[i];
36      app = tmp;
37    }
38    private void diminuisci() {
39      Appuntamento[] tmp = new Appuntamento[app.length / 2];
40      for (int i = 0; i < n; i++)
41        tmp[i] = app[i];
42      app = tmp;
43    }
44    public String toString() {
45      String s = new String();
46      for (int i = 0; i < n; i++)
47        s += (i + ") " + app[i] +
48              System.getProperty("line.separator"));
49      return s;
50    }
51  }
```

Operaz.java:

```
1   package cap15.app;
2
3   enum Operaz {
4     dammi, salva, esci;
5   }
```

InterfUtente.java:

```
1   package cap15.app;
2
3   import java.io.*;
4   import java.net.*;
5   import java.util.*;
6   public class InterfUtente {
7     private static String DEFINDIR = "localhost";
8     private static int DEFPORTA = 2000;
9     private String indir;
10    private int porta;
11    private Socket s;
12    private String nomeUtente;
13    private Agenda agenda;
```

```
14    private Scanner tas = new Scanner(System.in);
15    private ObjectInputStream ois;
16    private ObjectOutputStream oos;
17    public InterfUtente() {
18      this(DEFINDIR, DEFPORTA);
19    }
20    public InterfUtente(String i) {
21      this(i, DEFPORTA);
22    }
23    public InterfUtente(String i, int p) {
24      indir = i;
25      porta = p;
26    }
27    private void connetti() throws IOException,
28      ClassNotFoundException {
29      System.out.println("Come ti chiami?");
30      nomeUtente = tas.next();
31      s = new Socket(indir, porta);
32      oos = new ObjectOutputStream(s.getOutputStream());
33      ois = new ObjectInputStream(s.getInputStream());
34    }
35    private void login() throws IOException,
36      ClassNotFoundException {
37      oos.writeObject(Operaz.dammi);
38      oos.writeObject(nomeUtente);
39      agenda = (Agenda) ois.readObject();
40    }
41    private void menu() {
42      System.out.println("Appuntamenti:");
43      System.out.println(agenda);
44      System.out.println("a) aggiungi un appuntamento");
45      System.out.println("b) rimuovi un appuntamento");
46      System.out.println("c) salva sul server");
47      System.out.println("d) esci");
48    }
49    public void opera() {
50      boolean vai = true;
51      char c;
52      try {
53        connetti();
54        login();
```

```
55          while (vai) {
56            menu();
57            c = tas.next().charAt(0);
58            switch (c) {
59            case 'a':
60              aggiungi();
61              break;
62            case 'b':
63              rimuovi();
64              break;
65            case 'c':
66              salva();
67              break;
68            case 'd':
69              esci();
70              vai = false;
71            }
72          }
73        } catch (IOException ioe) {
74          ioe.printStackTrace();
75        } catch (ClassNotFoundException cnfe) {
76          cnfe.printStackTrace();
77        }
78      }
79
80      private void aggiungi() {
81        Data d;
82        while (true) {
83          System.out.println("Inserisci la data (g/m/a)");
84          try {
85            d = new Data(tas.next());
86            break;
87          } catch (Exception e) {
88          }
89        }
90        System.out.println("Inserisci il titolo");
91        tas.useDelimiter("\n");
92        String t = tas.next();
93        int o;
94        while (true) {
95          System.out.println("A che ore?");
```

```
96          try {
97            o = tas.nextInt();
98            break;
99          } catch (InputMismatchException e) {
100           tas.next();
101         }
102       }
103       Appuntamento ap = new Appuntamento(t, d, o);
104       agenda.inserisci(ap);
105     }
106
107     private void rimuovi() {
108       while (true) {
109         try {
110           System.out.println("Quale vuoi rimuovere?");
111           int q = tas.nextInt();
112           agenda.rimuovi(q);
113           break;
114         } catch (InputMismatchException e) {
115           System.out.println("Indice non valido");
116           tas.next();
117         } catch (IndiceNonValidoException e) {
118           System.out.println("Indice non valido");
119         }
120       }
121     }
122
123     private void salva() throws IOException {
124       oos.writeObject(Operaz.salva);
125       oos.writeObject(nomeUtente);
126       oos.writeObject(agenda);
127     }
128
129     private void esci() throws IOException {
130       oos.writeObject(Operaz.esci);
131     }
132
133     public static void main(String[] args) {
134       InterfUtente iu;
135       switch (args.length) {
136       case 0:
```

```
137         iu = new InterfUtente();
138         break;
139       case 1:
140         iu = new InterfUtente(args[0]);
141         break;
142       case 2:
143         iu = new InterfUtente(args[0],
144                               Integer.parseInt(args[1]));
145         break;
146       default:
147         System.out.println("Uso: java InterfUtente" +
148                            "<indirizzoserver> <porta>");
149         return;
150       }
151       iu.opera();
152     }
153   }
```

Server.java:

```
1   package cap15.app;
2
3   import java.io.*;
4   import java.net.*;
5   public class Server {
6     private static int DEFPORTA = 2000;
7     private ServerSocket srvs;
8     public Server() throws IOException {
9       this(DEFPORTA);
10    }
11    public Server(int p) throws IOException {
12      srvs = new ServerSocket(p);
13    }
14    public void vai() throws IOException,
15      ClassNotFoundException {
16      ObjectInputStream ois;
17      ObjectOutputStream oos;
18      while (true) {
19        Socket s = srvs.accept();
20        ois = new ObjectInputStream(s.getInputStream());
21        oos = new ObjectOutputStream(s.getOutputStream());
22        boolean ancora = true;
```

```
23          while (ancora) {
24            Operaz oo = (Operaz) ois.readObject();
25            switch (oo) {
26            case dammi:
27              String nome = (String) ois.readObject();
28              Agenda a = leggi(nome);
29              oos.writeObject(a);
30              break;
31            case salva:
32              String no = (String) ois.readObject();
33              Agenda aa = (Agenda) ois.readObject();
34              salva(no, aa);
35              break;
36            case esci:
37              ancora = false;
38            }
39          }
40        }
41      }
42      private Agenda leggi(String n) throws IOException,
43        ClassNotFoundException {
44        File f = new File(n);
45        if (!f.exists()) {
46          return new Agenda();
47        }
48        ObjectInputStream fois = new ObjectInputStream(
49          new FileInputStream(n));
50        Agenda a = (Agenda) fois.readObject();
51        fois.close();
52        return a;
53      }
54      private void salva(String n,
55                         Agenda a) throws IOException {
56        ObjectOutputStream foos = new ObjectOutputStream(
57          new FileOutputStream(n));
58        foos.writeObject(a);
59        foos.close();
60      }
61      public static void main(String[] args)
62      throws IOException, ClassNotFoundException {
63        Server server = new Server();
```

```
64          server.vai();
65      }
66  }
```

16. Invocazione di metodi remoti

16.1 Numeri primi

Realizzare un programma servitore che esporta un oggetto remoto dotato del seguente metodo:

int[] calcolaPrimi(int q) throws RemoteException

All'invocazione del metodo, vengono calcolati i primi *q* numeri primi e vengono restituiti al chiamante. Il servitore provvede a pubblicizzare il servizio esportato dall'oggetto remoto attraverso il registro RMI. Realizzare inoltre un programma cliente che: *i*) si procura un riferimento dell'oggetto remoto attraverso il registro RMI; *ii*) invoca il metodo passando un argomento attuale scelto dall'utente; *iii*) stampa a video i risultati ottenuti.

Quindi eseguire il cliente ed il servitore in due cartelle separate, in maniera simile a quanto avviene nel caso in cui i componenti della applicazione siano effettivamente distribuiti (per semplicità non si consideri il trasferimento dinamico della classe stub).

Soluzione

NumPrimi.java:

```
1  package cap16.numprimi;
2
3  import java.rmi.*;
4  public interface NumPrimi extends Remote {
5    public int[] calcolaPrimi(int q) throws
6      RemoteException;
```

```
7   }
```

NumPrimiImpl.java:

```
1   package cap16.numprimi;
2
3   import java.rmi.*;
4   import java.rmi.server.*;
5   public class NumPrimiImpl extends UnicastRemoteObject
6     implements NumPrimi {
7     public NumPrimiImpl() throws RemoteException {
8     }
9     public int[] calcolaPrimi(int q) throws
10      RemoteException {
11      int[] risul = new int[q];
12      int n = 0;
13      int i = 2;
14      while (n < q) {
15        boolean primo = true;
16        for (int d = 2; d <= Math.sqrt(i); d++)
17          if ((i % d) == 0) {
18            primo = false;
19            break;
20          }
21        if (primo) {
22          risul[n++] = i;
23        }
24        i++;
25      }
26      return risul;
27    }
28  }
```

Servitore.java:

```
1   package cap16.numprimi;
2
3   import java.net.*;
4   import java.rmi.*;
5   public class Servitore {
6     public static void main(String[] args) {
7       try {
```

```
8         NumPrimi np = new NumPrimiImpl();
9         Naming.rebind("//localhost/numprimi", np);
10      } catch (RemoteException re) {
11        System.err.println("Errore di rete");
12      } catch (MalformedURLException mue) {
13        System.err.println(
14          "Errore durante l'operazione rebind");
15      }
16    }
17  }
```

Cliente.java:

```
1   package cap16.numprimi;
2
3   import java.net.*;
4   import java.rmi.*;
5   public class Cliente {
6     public static void main(String[] args) {
7       if (args.length != 2) {
8         System.err.println(
9           "Uso: java cap16.numprimi.Cliente nomehost numero");
10        System.exit(1);
11      }
12      try {
13        NumPrimi np =
14          (NumPrimi) Naming.lookup(
15            "//" + args[0] + "/numprimi");
16        int n = Integer.parseInt(args[1]);
17        int[] risul = np.calcolaPrimi(n);
18        for (int i = 0; i < risul.length; i++)
19          System.out.print(risul[i] + " ");
20        System.out.println();
21      } catch (NumberFormatException nfe) {
22        System.err.println(
23          "Parametro di ingresso non corretto");
24      } catch (NotBoundException nbe) {
25        System.err.println("Servizio non registrato");
26      } catch (MalformedURLException mue) {
27        System.err.println("URL non corretto");
28      } catch (RemoteException re) {
29        System.err.println("Eccezione di rete");
```

```
30            }
31        }
32    }
```

Note

L'implementazione del metodo *calcolaPrimi()* fornita dalla classe *NumPrimiImpl* non richiede la presenza di meccanismi atti a garantire la mutua esclusione. Il motivo risiede nel fatto che tutti gli oggetti e i riferimenti utilizzati sono allocati sullo stack, e pertanto l'invocazione simultanea del metodo da parte di più clienti non può portare il sistema in uno stato inconsistente.

Esecuzione

Supponiamo che i sorgenti siano contenuti nella cartella *src*, che le cartelle usate per simulare due host siano *cln* e *srv*, e che la macchina abbia nome *pc-frosini.iet.unipi.it*. Per compilare i sorgenti, dalla cartella *src* eseguire il seguente comando:

javac -d . *.java

L'opzione *-d .* fa sì che vengano create automaticamente le sottocartelle corrispondenti ai package di appartenenza delle varie classi (cioè *cap16/numprimi*). Quindi, per generare le classi stub e skeleton, eseguire il seguente comando:

rmic cap16.numprimi.NumPrimiImpl

Copiare nella cartella *cln/cap16/numprimi* i file *Cliente.class*, *NumPrimi.class*, e *NumPrimiImpl_Stub.class*, e nella cartella *srv/cap16/numprimi* i file *Servitore.class*, *NumPrimi.class*, *NumPrimiImpl.class*, e *NumPrimiImpl_Skel.class* (quest'ultima può essere omessa se si utilizza una piattaforma Java con numero di versione maggiore o uguale di 1.2).

Per attivare il registro RMI, portarsi nella cartella *srv* e eseguire il comando

rmiregistry

Il classpath associato al registro RMI è, se non specificato diversamente, pari alla cartella corrente, in questo modo il registro RMI è in grado di recuperare la definizione della classe stub e dell'interfaccia remota durante il processo di registrazione dell'oggetto remoto.

Per attivare il servitore, usando un altro interprete di comandi portarsi nella cartella *srv* e inviare il seguente comando:

java cap16.numprimi.Servitore

Infine, per attivare il cliente, usando un altro interprete di comandi portarsi nella cartella *cln* ed eseguire il seguente comando:

java cap16.numprimi.Cliente pc-frosini.iet.unipi.it 10

16.2 Teatro

Realizzare un programma servitore che gestisce i posti di un teatro. Il servitore esporta un oggetto remoto dotato dei metodi elencati di seguito.

- *int postiDisponibili() throws RemoteException*: restituisce il numero di posti ancora disponibili.
- *Prenotazione prenota(DatiCompratore dc) throws RemoteException, TeatroPienoException*: prenota uno dei posti disponibili assegnandolo al compratore passato come argomento e restituisce i dati della prenotazione; se non ci sono posti disponibili viene sollevata l'eccezione *TeatroPienoException*.
- *boolean rilascia(DatiCompratore dc, Prenotazione dp) throws RemoteException*: rende nuovamente disponibile il posto specificato da *dp* e precedentemente assegnato al compratore specificato da *dc*; restituisce true se l'operazione va a buon fine, false altrimenti.

Definire la classe *DatiCompratore* in modo tale che contenga nome, cognome e numero di telefono del compratore. Definire la classe *Prenotazione* in modo tale da memorizzare la fila e il numero del posto assegnato.

Realizzare infine un programma cliente che esegue alcune semplici operazioni sull'oggetto remoto.

Soluzione

DatiCompratore.java:

```
1  package cap16.teatro;
2
3  import java.io.*;
4  public class DatiCompratore implements Serializable {
5    private String nome;
6    private String cognome;
7    private String numTel;
8    public DatiCompratore(String n, String c, String t) {
9      nome = n;
10     cognome = c;
11     numTel = t;
12   }
13   public String getNome() {
14     return nome;
15   }
```

```
16    public String getCognome() {
17      return cognome;
18    }
19    public String getTel() {
20      return numTel;
21    }
22    public boolean equals(Object o) {
23      if (!(o instanceof DatiCompratore)) {
24        return false;
25      }
26      DatiCompratore d = (DatiCompratore) o;
27      return nome.equals(d.nome)
28             && cognome.equals(d.cognome) &&
29             numTel.equals(d.numTel);
30    }
31  }
```

Prenotazione.java:

```
1   package cap16.teatro;
2
3   import java.io.*;
4   public class Prenotazione implements Serializable {
5     private int fila;
6     private int numPosto;
7     public Prenotazione(int f, int n) {
8       fila = f;
9       numPosto = n;
10    }
11    public int getFila() {
12      return fila;
13    }
14    public int getNumPosto() {
15      return numPosto;
16    }
17    public boolean equals(Object o) {
18      if (!(o instanceof Prenotazione)) {
19        return false;
20      }
21      Prenotazione p = (Prenotazione) o;
22      return (fila == p.fila) && (numPosto == p.numPosto);
23    }
```

```
24    public String toString() {
25      return "fila=" + fila + ", posto=" + numPosto;
26    }
27  }
```

TeatroPienoException.java:

```
1   package cap16.teatro;
2
3   public class TeatroPienoException extends Exception {
4   }
```

Teatro.java:

```
1   package cap16.teatro;
2
3   import java.rmi.*;
4   public interface Teatro extends Remote {
5     public int postiDisponibili() throws RemoteException;
6     public Prenotazione prenota(DatiCompratore dc)
7     throws RemoteException, TeatroPienoException;
8     public boolean rilascia(DatiCompratore dc,
9                             Prenotazione dp)
10    throws RemoteException;
11  }
```

TeatroImpl.java:

```
1   package cap16.teatro;
2
3   import java.rmi.*;
4   import java.rmi.server.*;
5   public class TeatroImpl extends UnicastRemoteObject
6     implements Teatro {
7     DatiCompratore[][] t;
8     int disponibili;
9     public TeatroImpl(int nf,
10                      int np) throws RemoteException {
11      t = new DatiCompratore[nf][np];
12      disponibili = nf * np;
13    }
14    public int postiDisponibili() throws RemoteException {
15      return disponibili;
```

```
16    }
17    public synchronized Prenotazione prenota(
18      DatiCompratore dc)
19    throws RemoteException, TeatroPienoException {
20      for (int i = 0; i < t.length; i++)
21        for (int j = 0; j < t[0].length; j++)
22          if (t[i][j] == null) {
23            t[i][j] = dc;
24            return new Prenotazione(i, j);
25          }
26      throw new TeatroPienoException();
27    }
28    public synchronized boolean rilascia(DatiCompratore
29                                         dc,
30                                         Prenotazione p)
31    throws RemoteException {
32      int nf = p.getFila();
33      int np = p.getNumPosto();
34      if ((t[nf][np] != null) && t[nf][np].equals(dc)) {
35        t[nf][np] = null;
36        return true;
37      } else {
38        return false;
39      }
40    }
41  }
```

Servitore.java:

```
1   package cap16.teatro;
2
3   import java.net.*;
4   import java.rmi.*;
5   public class Servitore {
6     public static void main(String[] args) {
7       try {
8         Teatro tt = new TeatroImpl(3, 3);
9         Naming.rebind("//localhost/teatro", tt);
10      } catch (RemoteException re) {
11        System.err.println("Errore di rete");
12      } catch (MalformedURLException mue) {
13        System.err.println("URL errato");
```

```
14      }
15    }
16  }
```

Cliente.java:

```
1   package cap16.teatro;
2
3   import java.net.*;
4   import java.rmi.*;
5   public class Cliente {
6     DatiCompratore datiPersonali;
7     String urlServizio;
8     public Cliente(String n, String c, String t) {
9       datiPersonali = new DatiCompratore(n, c, t);
10    }
11    public void interagisci(String u) {
12      urlServizio = u;
13      try {
14        Teatro teatro = (Teatro) Naming.lookup(urlServizio);
15        System.out.println("Sono ancora disponibili " +
16                           teatro.postiDisponibili() + " posti");
17        Prenotazione p1 = teatro.prenota(datiPersonali);
18        System.out.println("Ho prenotato il seguente posto: "
19                           + p1);
20        Prenotazione p2 = teatro.prenota(datiPersonali);
21        System.out.println("Ho prenotato il seguente posto: "
22                           + p2);
23        System.out.println("Rilascio la prima prenotazione...");
24        System.out.println(teatro.rilascia(datiPersonali,
25                                           p1) ? "fatto" : "errore");
26      } catch (RemoteException re) {
27        System.err.println("Errore di rete");
28      } catch (MalformedURLException mue) {
29        System.err.println("URL errato");
30      } catch (NotBoundException nbe) {
31        System.err.println("Nessun oggetto con tale nome");
32      } catch (TeatroPienoException tpe) {
33        System.err.println("Il teatro e' pieno");
34      }
35    }
36    public static void main(String[] args) {
```

```
37      if (args.length != 4) {
38        System.out.println("Uso: java cap16.teatro.Cliente nome cognome tel host");
39        return;
40      }
41      Cliente c = new Cliente(args[0], args[1], args[2]);
42      c.interagisci("//" + args[3] + "/teatro");
43    }
44  }
```

16.3 Borsa

Un programma servitore rappresenta una borsa e tiene traccia dell'andamento del valore di un certo numero di titoli. I programmi clienti sono in grado di conoscere il valore di un titolo, impostare un nuovo valore per un titolo o registrarsi per ricevere notifiche quando il valore di un titolo supera una determinata soglia. Realizzare il servizio in modo tale che sia possibile eseguire su un oggetto di tale tipo le seguenti operazioni in remoto:

- *void setValore(String titolo, double v) throws BorsaException, RemoteException*: imposta il nuovo valore del titolo specificato; se il titolo non esiste viene sollevata una eccezione di tipo *BorsaException*.
- *double getValore(String titolo) throws BorsaException, RemoteException*: restituisce il valore del titolo specificato; se il titolo non esiste viene sollevata una eccezione di tipo *BorsaException*.
- *boolean registra(String titolo, double soglia, Notifica n) throws BorsaException, RemoteException*: registra l'oggetto *n* in modo tale che quest'ultimo venga notificato quando il titolo specificato ha un valore superiore alla soglia (callback); si supponga per semplicità che per ogni titolo possa registrarsi un solo cliente; il valore restituito dal metodo indica al chiamante se l'operazione di registrazione è andata a buon fine o meno; l'eccezione *BorsaException* viene sollevata se si tenta di registrarsi relativamente ad un titolo che non esiste.

Definire *Notifica* come una interfaccia remota dotata di un solo metodo (che viene invocato quando il valore supera la soglia). Dotare la classe *BorsaImpl* di un ulteriore metodo (locale) atto ad aggiungere titoli.

Realizzare quindi le classi dei programmi clienti che interagiscono con il sistema borsa: *Generatore* che varia il valore di un titolo usando i metodi *getValore()* e *setValore()*, e *Osservatore()* che si registra per ricevere le notifiche riguardanti un titolo attraverso il metodo *registra()*.

Soluzione

BorsaException.java:

```
1  package cap16.borsa;
2
3  public class BorsaException extends Exception {
4  }
```

Borsa.java:

```
1  package cap16.borsa;
2
3  import java.rmi.*;
4  public interface Borsa extends Remote {
5    public boolean registra(String titolo, double soglia,
6                            Notifica n)
7    throws BorsaException, RemoteException;
8    public void setValore(String titolo, double v)
9    throws BorsaException, RemoteException;
10   public double getValore(String titolo) throws
11     BorsaException, RemoteException;
12 }
```

Notifica.java:

```
1  package cap16.borsa;
2
3  import java.rmi.*;
4  public interface Notifica extends Remote {
5    public void notifica(String titolo, double valore)
6    throws RemoteException;
7  }
```

BorsaImpl.java:

```
1  package cap16.borsa;
2
3  import java.net.*;
4  import java.rmi.*;
5  import java.rmi.server.*;
6  public class BorsaImpl extends UnicastRemoteObject
7    implements Borsa {
```

```
8     private static int MAX_TITOLI = 10;
9     private Titolo[] tit;
10    private int numTitoli;
11    public BorsaImpl() throws RemoteException {
12      tit = new Titolo[MAX_TITOLI];
13    }
14    public synchronized boolean registra(String
15                                         nomeTitolo,
16                                         double soglia,
17                                         Notifica n) throws BorsaException, RemoteException
18      for (int i = 0; i < numTitoli; i++) {
19        if (tit[i].getNome().equals(nomeTitolo)) {
20          return tit[i].setCliente(n, soglia);
21        }
22      }
23      throw new BorsaException();
24    }
25    public synchronized void setValore(String t, double v)
26    throws BorsaException, RemoteException {
27      for (int i = 0; i < numTitoli; i++) {
28        if (tit[i].getNome().equals(t)) {
29          tit[i].setValore(v);
30          return;
31        }
32      }
33      throw new BorsaException();
34    }
35    public synchronized double getValore(String t)
36    throws BorsaException, RemoteException {
37      for (int i = 0; i < numTitoli; i++)
38        if (tit[i].getNome().equals(t)) {
39          return tit[i].getValore();
40        }
41      throw new BorsaException();
42    }
43    public synchronized boolean aggiungiTitolo(
44      String nome,
45      double valoreIniziale) {
46      if (numTitoli < MAX_TITOLI) {
47        tit[numTitoli] = new Titolo(nome, valoreIniziale);
48        numTitoli++;
```

```
49        return true;
50      }
51      return false;
52    }
53    public static void main(String[] args) {
54      try {
55        BorsaImpl bi = new BorsaImpl();
56        bi.aggiungiTitolo("Superplastic", 44.0);
57        bi.aggiungiTitolo("MacroSoft", 31.5);
58        bi.aggiungiTitolo("MoonMicrosystems", 39.7);
59        Naming.rebind("//localhost/borsa", bi);
60      } catch (RemoteException re) {
61        System.err.println("Errore di rete");
62      } catch (MalformedURLException mue) {
63        System.err.println("URL errato");
64      }
65    }
66    class Titolo {
67      private String nome;
68      private double valore;
69      private Notifica cliente;
70      private double soglia;
71
72      Titolo(String n, double v) {
73        nome = n;
74        valore = v;
75      }
76
77      double getValore() {
78        return valore;
79      }
80
81      void setValore(double v) {
82        valore = v;
83        if ((cliente != null) && (v >= soglia)) {
84          try {
85            cliente.notifica(nome, v);
86          } catch (RemoteException re) {
87            cliente = null;
88          }
89        }
```

```
90      }
91
92      String getNome() {
93        return nome;
94      }
95
96      boolean setCliente(Notifica n, double s) {
97        if (cliente == null) {
98          cliente = n;
99          soglia = s;
100         return true;
101       }
102       return false;
103     }
104   }
105 }
```

Osservatore.java:

```
1   package cap16.borsa;
2
3   import java.net.*;
4   import java.rmi.*;
5   import java.rmi.server.*;
6   public class Osservatore {
7     Borsa borsa;
8     public Osservatore(Borsa b) {
9       borsa = b;
10    }
11    public void osserva(String t, double s)
12    throws RemoteException, BorsaException {
13      borsa.registra(t, s, new NotificaImpl());
14    }
15    public static void main(String[] args) {
16      if (args.length != 3) {
17        System.err.println(
18          "Uso: java Osservatore url-servizio nome-titolo soglia");
19        System.exit(1);
20      }
21      try {
22        Borsa b = (Borsa) Naming.lookup(args[0]);
23        Osservatore o = new Osservatore(b);
```

```
24          o.osserva(args[1], Double.parseDouble(args[2]));
25        } catch (RemoteException re) {
26          System.err.println("Errore di rete");
27        } catch (MalformedURLException mue) {
28          System.err.println("URL errato");
29        } catch (NotBoundException nbe) {
30          System.err.println("Servizio non registrato");
31        } catch (BorsaException be) {
32          System.err.println("Titolo non presente");
33        }
34      }
35      class NotificaImpl extends UnicastRemoteObject
36        implements
37        Notifica {
38        NotificaImpl() throws RemoteException {
39        }
40        public void notifica(String titolo, double valore)
41        throws RemoteException {
42          System.out.println("Il titolo " + titolo +
43                             " ha superato la soglia e vale " + valore);
44        }
45      }
46    }
```

Generatore.java:

```
1   package cap16.borsa;
2
3   import java.net.*;
4   import java.rmi.*;
5   public class Generatore extends Thread {
6     private static double MAX_DELTA = 10.0;
7     private Borsa borsa;
8     private long periodo;
9     private String titolo;
10    public Generatore(Borsa b, String t) {
11      this(b, t, 1000L);
12    }
13    public Generatore(Borsa b, String t, long p) {
14      borsa = b;
15      periodo = p;
16      titolo = t;
```

```
17      start();
18    }
19    public void run() {
20      try {
21        while (true) {
22          Thread.sleep(periodo);
23          double delta = ((2 * Math.random()) - 1.0) *
24                         MAX_DELTA;
25          double vecchioVal = borsa.getValore(titolo);
26          double nuovoVal = (((vecchioVal + delta) >= 0.0) ?
27                             (vecchioVal + delta)
28                             : 0.0);
29          borsa.setValore(titolo, nuovoVal);
30        }
31      } catch (InterruptedException ie) {
32      } catch (RemoteException re) {
33        System.err.println("Errore di rete");
34      } catch (BorsaException be) {
35        System.err.println("Titolo non presente");
36      }
37    }
38    public static void main(String[] args) {
39      if (args.length != 2) {
40        System.out.println("Uso: java cap16.borsa.Generatore urlserv titolo");
41        System.exit(1);
42      }
43      try {
44        Borsa b = (Borsa) Naming.lookup(args[0]);
45        Generatore g = new Generatore(b, args[1]);
46      } catch (RemoteException re) {
47        System.err.println("Errore di rete");
48      } catch (MalformedURLException mue) {
49        System.err.println("URL errato");
50      } catch (NotBoundException nbe) {
51        System.err.println("Servizio non registrato");
52      }
53    }
54  }
```

16.4 Esecuzione remota

Un servitore rende disponibili servizi di calcolo che possono essere utilizzati dai clienti. L'accesso ai servizi di calcolo avviene invocando il seguente metodo remoto, esportato dal servitore:

Object esegui(Compito c) throws RemoteException

dove *c* è un oggetto che indica il procedimento di calcolo e i dati su cui operare, mentre il valore restituito contiene il risultato del procedimento. La classe *Compito* è definita come segue:

Compito.java:

```
1  package cap16.er;
2
3  import java.io.*;
4  public abstract class Compito implements
5    Serializable {
6    Object datiIniziali;
7    public void setDatiIniziali(Object o) {
8      datiIniziali = o;
9    }
10   public abstract Object calcola();
11 }
```

I programmi clienti possono definire dei nuovi compiti di calcolo come istanze di opportune sottoclassi di *Compito*, ed inviarli al servitore attraverso il metodo *esegui()*.

Realizzare il programma servitore ed un cliente che utilizza i servizi di calcolo per ordinare un vettore di interi. Eseguire l'applicazione sfruttando i meccanismi per il caricamento dinamico delle classi di RMI.

Soluzione

Esecutore.java:

```
1  package cap16.er;
2
3  import java.rmi.*;
4  public interface Esecutore extends Remote {
5    public Object esegui(Compito c) throws
6      RemoteException;
7  }
```

EsecutoreImpl.java:

```
1  package cap16.er;
2
3  import java.rmi.*;
4  import java.rmi.server.*;
5  public class EsecutoreImpl extends UnicastRemoteObject
6    implements Esecutore {
7    public EsecutoreImpl() throws RemoteException {}
8    public Object esegui(Compito c) throws
9      RemoteException {
10     return c.calcola();
11   }
12 }
```

Servitore.java:

```
1  package cap16.er;
2
3  import java.net.*;
4  import java.rmi.*;
5  public class Servitore {
6    public static void main(String[] args) {
7      try {
8        Esecutore e = new EsecutoreImpl();
9        System.err.println("Oggetto creato");
10       Naming.rebind("//localhost/er", e);
11     } catch (RemoteException re) {
12       System.err.println("Errore di rete");
13     } catch (MalformedURLException mue) {
14       System.err.println("URL errato");
15     }
16   }
17 }
```

Ordinamento.java:

```
1  package cap16.er;
2
3  public class Ordinamento extends Compito {
4    public Object calcola() {
5      int[] vettore = (int[]) datiIniziali;
6      int n = vettore.length;
7      boolean ordinato = false;
```

```
8       for (int i = 0; (i < (n - 1)) && !ordinato; i++) {
9         ordinato = true;
10        for (int j = n - 1; j >= (i + 1); j--)
11          if (vettore[j] < vettore[j - 1]) {
12            int tmp = vettore[j];
13            vettore[j] = vettore[j - 1];
14            vettore[j - 1] = tmp;
15            ordinato = false;
16          }
17      }
18      return vettore;
19    }
20  }
```

Cliente.java:

```
1   package cap16.er;
2
3   import java.net.*;
4   import java.rmi.*;
5   public class Cliente {
6     public static void main(String[] args) {
7       try {
8         Esecutore e = (Esecutore)
9                       Naming.lookup("//localhost/er");
10        Ordinamento o = new Ordinamento();
11        int[] v = new int[] { 1, 9, 4, 7, 55, 44, 21, 6 };
12        o.setDatiIniziali(v);
13        int[] risultato = (int[]) e.esegui(o);
14        for (int i = 0; i < v.length; i++)
15          System.out.print(risultato[i] + " ");
16        System.out.println();
17      } catch (RemoteException re) {
18        System.err.println("Errore di rete");
19      } catch (NotBoundException nbe) {
20        System.err.println("Servizio non registrato");
21      } catch (MalformedURLException mue) {
22        System.err.println("URL errato");
23      }
24    }
25  }
```

17. Interfacce grafiche

17.1 Giorni

Scrivere un programma, dotato di una interfaccia simile a quella mostrata in Figura 17.1, che chiede all'utente di inserire la data di nascita e calcola il numero di giorni trascorsi.

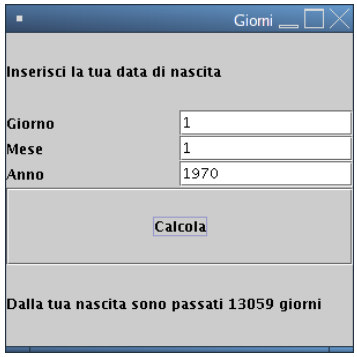


Figura 17.1: Quando l'utente preme il bottone "Calcola", viene prelevato il valore delle tre aree di testo e il numero di giorni trascorsi viene visualizzato mediante l'etichetta posta in basso.

Suggerimenti

Una data può essere rappresentata attraverso una istanza della classe *GregorianCalendar* (package *java.util*). La classe dispone di numerosi costruttori e metodi tra cui:

- ***GregorianCalendar()***: crea una istanza usando la data e l'ora corrente.
- ***GregorianCalendar(int a, int m, int g)***: crea una istanza relativa al giorno *g*, del mese *m*, dell'anno *a* (i mesi sono specificati con valori che partono da zero, per esempio gennaio=0, febbraio=1, etc.).
- ***long getTimeInMillis()***: restituisce il numero di millisecondi trascorsi dall'ora 00:00:00.0000 del primo gennaio 1970.

Soluzione

Giorni.java:

```
1  package cap17.giorni;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.util.*;
6  import javax.swing.*;
7  public class Giorni extends JFrame {
8    JLabel lab1 = new
9    JLabel("Inserisci la tua data di nascita");
10   JTextField giorno = new JTextField("1");
11   JTextField mese = new JTextField("1");
12   JTextField anno = new JTextField("1970");
13   JLabel glab = new JLabel("Giorno");
14   JLabel mlab = new JLabel("Mese");
15   JLabel alab = new JLabel("Anno");
16   JButton calcola = new JButton("Calcola");
17   JLabel lab2 = new JLabel();
18   GridLayout gridLayout1 = new GridLayout(3, 2);
19   GridLayout gridLayout2 = new GridLayout(4, 0);
20   JPanel pan1 = new JPanel();
21   public Giorni() {
22     super("Giorni");
23     pan1.setLayout(gridLayout1);
24     pan1.add(glab);
25     pan1.add(giorno);
26     pan1.add(mlab);
27     pan1.add(mese);
28     pan1.add(alab);
29     pan1.add(anno);
```

```
30      Container c = getContentPane();
31      c.setLayout(gridLayout2);
32      c.add(lab1);
33      c.add(pan1);
34      c.add(calcola);
35      c.add(lab2);
36      calcola.addActionListener(new AscoltatoreBot());
37      setDefaultCloseOperation(DISPOSE_ON_CLOSE);
38    }
39    static boolean dataValida(int g, int m, int a) {
40      if ((m < 1) || (m > 12) || (a < 1583)) {
41        return false;
42      }
43      int gg;
44      switch (m) {
45      case 4:
46      case 6:
47      case 9:
48      case 11:
49        gg = 30;
50        break;
51      case 2:
52        if (((a % 4) != 0) || (((a % 100) == 0)
53                               && ((a % 400) != 0))) {
54          gg = 28;
55        } else {
56          gg = 29;
57        }
58        break;
59      default:
60        gg = 31;
61      }
62      return (g > 0) && (g <= gg);
63    }
64    public static void main(String[] args) {
65      Giorni g = new Giorni();
66      g.setSize(300, 300);
67      g.setVisible(true);
68    }
69    class AscoltatoreBot implements ActionListener {
70      public void actionPerformed(ActionEvent ae) {
```

```
71          try {
72            int g = Integer.parseInt(giorno.getText());
73            int m = Integer.parseInt(mese.getText());
74            int a = Integer.parseInt(anno.getText());
75            if (dataValida(g, m, a)) {
76              GregorianCalendar now = new GregorianCalendar();
77              GregorianCalendar x = new GregorianCalendar(a, m - 1,
78                  g);
79              long ms = now.getTimeInMillis() - x.getTimeInMillis();
80              long gg = ms / (24 * 60 * 60 * 1000);
81              lab2.setText("Dalla tua nascita sono passati " + gg +
82                           " giorni");
83            } else {
84              lab2.setText("Data non valida");
85            }
86          } catch (NumberFormatException nfe) {
87            lab2.setText("Controlla i valori immessi");
88          }
89        }
90      }
91    }
```

17.2 Editor

Realizzare un semplice editor di testi simile a quello mostrato in Figura 17.2.

Suggerimenti

La classe *JTextArea* dispone dei seguenti metodi:

- ***String getText()***: restituisce il testo contenuto nell'area.
- ***void setText(String s)***: cambia il testo contenuto nell'area.
- ***void append(String s)***: appende la stringa specificata in fondo al testo.
- ***String getSelectedText()***: restituisce il testo selezionato.
- ***void replaceSelection(String s)***: sostituisce il testo selezionato con quello specificato come argomento.
- ***int getCaretPosition()***: restituisce la posizione del cursore.
- ***void setCaretPosition()***: sposta il cursore nella posizione indicata.

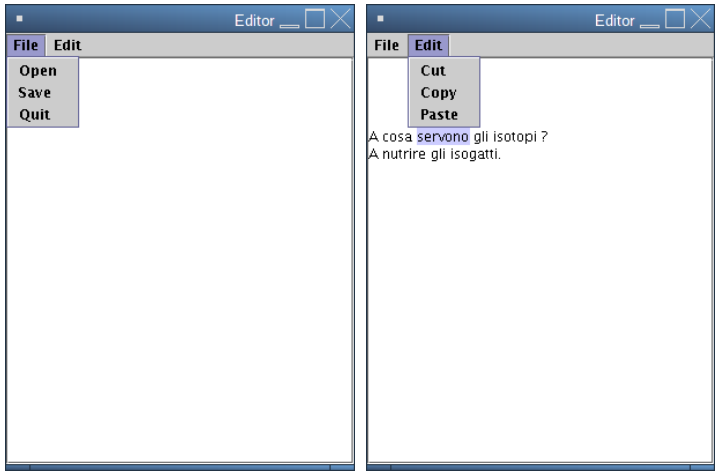


Figura 17.2: Un semplice editor di testi

Soluzione

Editor.java:

```
1   package cap17.editor;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.io.*;
6   import javax.swing.*;
7   public class Editor extends JFrame {
8     JMenuBar bar = new JMenuBar();
9     JMenu fileMenu = new JMenu("File");
10    JMenu editMenu = new JMenu("Edit");
11    JMenuItem open = new JMenuItem("Open");
12    JMenuItem save = new JMenuItem("Save");
13    JMenuItem quit = new JMenuItem("Quit");
14    JMenuItem cut = new JMenuItem("Cut");
15    JMenuItem copy = new JMenuItem("Copy");
16    JMenuItem paste = new JMenuItem("Paste");
17    JTextArea area = new JTextArea();
18    JScrollPane scrPane = new JScrollPane(area);
19    String buffer;
20    public Editor() {
```

```
21      super("Editor");
22      fileMenu.add(open);
23      fileMenu.add(save);
24      fileMenu.add(quit);
25      editMenu.add(cut);
26      editMenu.add(copy);
27      editMenu.add(paste);
28      bar.add(fileMenu);
29      bar.add(editMenu);
30      AscoltatoreMenu1 fm = new AscoltatoreMenu1();
31      open.addActionListener(fm);
32      save.addActionListener(fm);
33      quit.addActionListener(fm);
34      AscoltatoreMenu2 em = new AscoltatoreMenu2();
35      cut.addActionListener(em);
36      copy.addActionListener(em);
37      paste.addActionListener(em);
38      setJMenuBar(bar);
39      getContentPane().add(scrPane);
40    }
41    public static void main(String[] args) {
42      Editor e = new Editor();
43      e.setSize(300, 400);
44      e.setVisible(true);
45      e.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
46    }
47    class AscoltatoreMenu1 implements ActionListener {
48      public void actionPerformed(ActionEvent e) {
49        JMenuItem mi = (JMenuItem) e.getSource();
50        if (mi == open) {
51          JFileChooser fc = new JFileChooser();
52          int ret = fc.showOpenDialog(Editor.this);
53          if (ret == JFileChooser.APPROVE_OPTION) {
54            File file = fc.getSelectedFile();
55            try {
56              BufferedReader br =
57                new BufferedReader(new FileReader(file));
58              String s;
59              area.setText("");
60              while ((s = br.readLine()) != null)
61                area.append(s + "\n");
```

```
62              br.close();
63            } catch (IOException ioe) {
64              JOptionPane.showMessageDialog(
65                Editor.this,
66                "Non riesco a leggere/aprire il file",
67                "Errore", JOptionPane.ERROR_MESSAGE);
68            }
69          }
70        } else if (mi == save) {
71          JFileChooser fc = new JFileChooser();
72          int ret = fc.showOpenDialog(Editor.this);
73          if (ret == JFileChooser.APPROVE_OPTION) {
74            File file = fc.getSelectedFile();
75            try {
76              BufferedWriter br =
77                new BufferedWriter(new FileWriter(file));
78              String s = area.getText();
79              br.write(s, 0, s.length());
80              br.close();
81            } catch (IOException ioe) {
82              JOptionPane.showMessageDialog(
83                Editor.this, "Non riesco a scrivere nel file",
84                "Errore", JOptionPane.ERROR_MESSAGE);
85            }
86          }
87        } else if (mi == quit)
88          System.exit(0);
89      }
90    }
91    class AscoltatoreMenu2 implements ActionListener {
92      public void actionPerformed(ActionEvent e) {
93        JMenuItem mi = (JMenuItem) e.getSource();
94        if (mi == cut) {
95          buffer = area.getSelectedText();
96          area.replaceSelection("");
97          area.setCaretPosition(area.getCaretPosition());
98        } else if (mi == copy) {
99          buffer = area.getSelectedText();
100         area.setCaretPosition(area.getCaretPosition());
101       } else if (mi == paste) {
102         area.replaceSelection(buffer);
```

```
103             area.setCaretPosition(area.getCaretPosition());
104           }
105         }
106       }
107     }
```

17.3 Calcolatrice

Si scriva un programma che realizza una calcolatrice in grado di operare su numeri interi simile a quella mostrata in Figura 17.3.

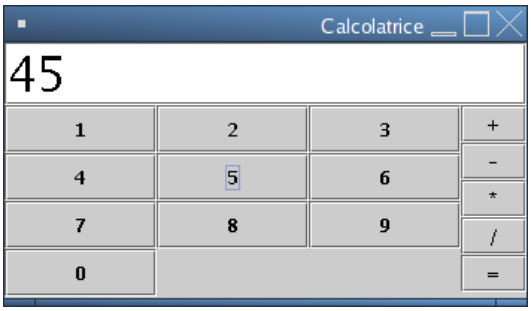


Figura 17.3: Aspetto della calcolatrice

Soluzione

Calcolatrice.java:

```
1   package cap17.calcolatrice;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import javax.swing.*;
6   public class Calcolatrice extends JFrame {
7     JTextField disp;
8     JPanel pad;
9     JPanel ops;
10    JButton[] b = new JButton[10];
11    JButton sum;
12    JButton div;
13    JButton mul;
```

```
14      JButton sub;
15      JButton ug;
16      int accumul;
17      NumListener numListener;
18      OpListener opListener;
19      boolean primo = true;
20      Op op = Op.NO;
21      public Calcolatrice() {
22        super("Calcolatrice");
23        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24        disp = new JTextField("0");
25        disp.setFont(new Font("Arial", Font.PLAIN, 32));
26        pad = new JPanel();
27        pad.setLayout(new GridLayout(4, 3));
28        ops = new JPanel();
29        numListener = new NumListener();
30        opListener = new OpListener();
31        for (int i = 1; i < 10; i++) {
32          b[i] = new JButton(String.valueOf(i));
33          pad.add(b[i]);
34          b[i].addActionListener(numListener);
35        }
36        b[0] = new JButton("0");
37        pad.add(b[0]);
38        b[0].addActionListener(numListener);
39        sum = new JButton("+");
40        sum.addActionListener(opListener);
41        sub = new JButton("-");
42        sub.addActionListener(opListener);
43        mul = new JButton("*");
44        mul.addActionListener(opListener);
45        div = new JButton("/");
46        div.addActionListener(opListener);
47        ug = new JButton("=");
48        ug.addActionListener(opListener);
49        ops.setLayout(new GridLayout(5, 1));
50        ops.add(sum);
51        ops.add(sub);
52        ops.add(mul);
53        ops.add(div);
54        ops.add(ug);
```

```
55      getContentPane().add("North", disp);
56      getContentPane().add("Center", pad);
57      getContentPane().add("East", ops);
58      setSize(350, 200);
59      setVisible(true);
60    }
61    public static void main(String[] args) {
62      Calcolatrice calc = new Calcolatrice();
63    }
64    enum Op {
65      NO, SUM, SUB, MUL, DIV;
66    }
67    class NumListener implements ActionListener {
68      public void actionPerformed(ActionEvent e) {
69        JButton button = (JButton) e.getSource();
70        String s = button.getText();
71        String tmp = disp.getText();
72        if (primo)
73          tmp = "";
74        primo = false;
75        tmp += s;
76        disp.setText(tmp);
77      }
78    }
79    class OpListener implements ActionListener {
80      public void actionPerformed(ActionEvent e) {
81        JButton button = (JButton) e.getSource();
82        String label = button.getText();
83        String n = disp.getText();
84        try {
85          int secOp = Integer.parseInt(n);
86          switch (op) {
87          case NO:
88            accumul = secOp;
89            break;
90          case SUM:
91            accumul += secOp;
92            break;
93          case SUB:
94            accumul -= secOp;
95            break;
```

```
96            case MUL:
97              accumul *= secOp;
98              break;
99            case DIV:
100             accumul /= secOp;
101             break;
102           }
103           disp.setText(String.valueOf(accumul));
104         } catch (ArithmeticException ae) {
105           disp.setText("Error");
106         } catch (NumberFormatException nfe) {
107           return;
108         } finally {
109           primo = true;
110         }
111         if (button == sum) {
112           op = Op.SUM;
113           return;
114         }
115         if (button == sub) {
116           op = Op.SUB;
117           return;
118         }
119         if (button == mul) {
120           op = Op.MUL;
121           return;
122         }
123         if (button == div) {
124           op = Op.DIV;
125           return;
126         }
127         if (button == ug) {
128           op = Op.NO;
129           return;
130         }
131       }
132     }
133   }
```

17.4 TicTacToe

Si scriva un programma che visualizza una finestra simile a quella mostrata in Figura 17.4. Il pannello della finestra contiene una griglia formata da N righe e N colonne. Ogni elemento della griglia può essere libero oppure occupato da un simbolo, ed i possibili simboli sono un cerchio ed una croce. Quando viene eseguito un click su un punto del pannello, l'elemento corrispondente della griglia, se libero, viene riempito con un simbolo (alternativamente con un cerchio ed una croce). Quando la griglia contiene una sequenza di N simboli uguali disposti su una riga, una colonna o una delle diagonali la partita termina e viene mostrato un opportuno messaggio.

La finestra è anche dotata di un menu che consente di salvare lo stato del gioco e di ripristinarlo successivamente. Si suggerisce di riutilizzare la classe *Tavola* (es. 12.1).

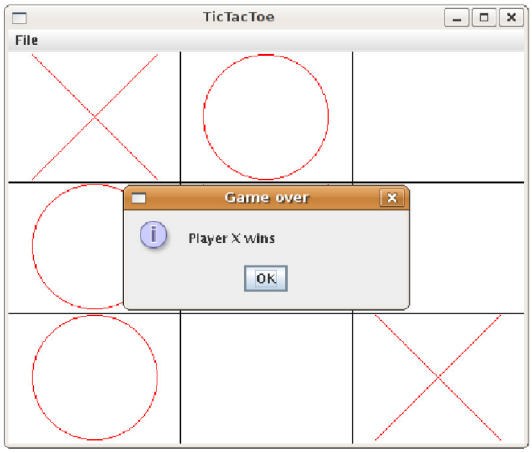


Figura 17.4: Aspetto della griglia e messaggio visualizzato al termine della partita

Soluzione

TicTacToe.java:

```
1  package cap17.tictactoe;
2
3  import cap12.tavola.Tavola;
4  import java.awt.*;
5  import java.awt.event.*;
```

```
6   import java.io.*;
7   import javax.swing.*;
8   public class TicTacToe extends JFrame {
9     class BoardPanel extends JPanel {
10      int W;
11      int H;
12      int N;
13      BoardPanel(int n) {
14        N = n;
15        tav = new Tavola(N);
16        init();
17        setBackground(Color.white);
18        addMouseListener(new AscoltatoreMouse());
19      }
20      void init() {
21        tav.svuota();
22        repaint();
23      }
24      void checkWinner() {
25        if (tav.vinceCroce()) {
26          JOptionPane.showMessageDialog(
27            this, "Player X wins", "Game over",
28            JOptionPane.INFORMATION_MESSAGE);
29          init();
30          return;
31        }
32        if (tav.vinceCerchio()) {
33          JOptionPane.showMessageDialog(
34            this, "Player O wins", "Game over",
35            JOptionPane.INFORMATION_MESSAGE);
36          init();
37          return;
38        }
39        if (tav.piena()) {
40          JOptionPane.showMessageDialog(
41            this, "There is no winner", "Game over",
42            JOptionPane.INFORMATION_MESSAGE);
43          init();
44        }
45      }
46      private void drawX(Graphics g, int x, int y) {
```

```
47        int s = ((((W < H) ? W : H) / N / 2) - 2);
48        s = (s < 2) ? 2 : s;
49        g.drawLine(x - s, y - s, x + s, y + s);
50        g.drawLine(x - s, y + s, x + s, y - s);
51      }
52      private void drawO(Graphics g, int x, int y) {
53        int s = ((((W < H) ? W : H) / N / 2) - 2);
54        s = (s < 2) ? 2 : s;
55        g.drawOval(x - s, y - s, 2 * s, 2 * s);
56      }
57      private void drawBoard(Graphics g) {
58        g.setColor(Color.black);
59        for (int i = 1; i < N; i++)
60          g.drawLine((i * W) / N, 0, (i * W) / N, H);
61        for (int i = 1; i < N; i++)
62          g.drawLine(0, (i * H) / N, W, (i * H) / N);
63      }
64      private void drawSimbols(Graphics g) {
65        g.setColor(Color.red);
66        for (int i = 0; i < N; i++)
67          for (int j = 0; j < N; j++)
68            if (tav.elemento(i, j) == Tavola.Stato.CRO)
69              drawX(
70                g, ((j * W) / N) + (W / N / 2),
71                ((i * H) / N) + (H / N / 2));
72            else if (tav.elemento(i, j) == Tavola.Stato.CER)
73              drawO(
74                g, ((j * W) / N) + (W / N / 2),
75                ((i * H) / N) + (H / N / 2));
76      }
77      public void paintComponent(Graphics g) {
78        super.paintComponent(g);
79        H = getHeight();
80        W = getWidth();
81        drawBoard(g);
82        drawSimbols(g);
83      }
84      class AscoltatoreMouse implements MouseListener {
85        public void mouseClicked(MouseEvent me) {
86          int x = me.getX();
87          int y = me.getY();
```

```
88          int j = (int) ((float) x / W * N);
89          int i = (int) ((float) y / H * N);
90          if (tav.elemento(i, j) == Tavola.Stato.LIBERA) {
91            tav.contrassegna(i, j);
92            repaint();
93            checkWinner();
94          }
95        }
96        public void mouseEntered(MouseEvent me) {}
97        public void mouseExited(MouseEvent me) {}
98        public void mousePressed(MouseEvent me) {}
99        public void mouseReleased(MouseEvent me) {}
100     }
101   }
102   class AscoltatoreMenu implements ActionListener {
103     public void actionPerformed(ActionEvent e) {
104       JMenuItem mi = (JMenuItem) e.getSource();
105       try {
106         if (mi == open) {
107           tav = Tavola.carica(filename);
108           repaint();
109         } else if (mi == save)
110           tav.salva(filename);
111         else if (mi == quit)
112           System.exit(0);
113       } catch (Exception ee) {
114         ee.printStackTrace();
115         JOptionPane.showMessageDialog(
116           TicTacToe.this, "File error", "File error",
117           JOptionPane.ERROR_MESSAGE);
118       }
119     }
120   }
121   Tavola tav;
122   String filename = "status.dat";
123   JMenuBar bar = new JMenuBar();
124   JMenu file = new JMenu("File");
125   JMenuItem open = new JMenuItem("Open");
126   JMenuItem save = new JMenuItem("Save");
127   JMenuItem quit = new JMenuItem("Quit");
128   AscoltatoreMenu am = new AscoltatoreMenu();
```

```
129     BoardPanel bp = new BoardPanel(3);
130     public TicTacToe() {
131       super("TicTacToe");
132       file.add(open);
133       file.add(save);
134       file.add(quit);
135       bar.add(file);
136       open.addActionListener(am);
137       save.addActionListener(am);
138       quit.addActionListener(am);
139       setJMenuBar(bar);
140       getContentPane().add(bp);
141     }
142     public static void main(String[] args) {
143       TicTacToe t = new TicTacToe();
144       t.setSize(200, 200);
145       t.setVisible(true);
146       t.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
147     }
148   }
```

17.5 Supercar

Vogliamo realizzare un programma che visualizza una animazione ciclica il cui aspetto, ad un dato istante, è simile a quello fotografato in Figura 17.5. Supponiamo di assegnare un indice ai quadrati, da 0 (quello più a sinistra) a N-1 (quello più a destra). Uno dei quadrati è più luminoso degli altri. Il quadrato più luminoso è inizialmente quello di indice 0. Successivamente il quadrato più luminoso diventa quello di indice 1 e così via fino ad arrivare a quello di indice N-1. A questo punto la direzione si inverte: il quadrato più luminoso diventa quello di indice N-2 e così via fino a tornare a quello di indice 0, dove il ciclo ricomincia.

Soluzione

Supercar.java:

```
1   package cap17.supercar;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import javax.swing.*;
```

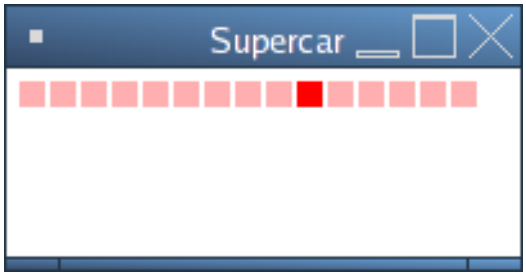


Figura 17.5: Aspetto della finestra a un dato istante

```
6   public class Supercar extends JPanel {
7     static final int SIZE = 10;
8     static final int SPACE = 12;
9     static final int BORDER = 5;
10    static final int WAIT = 300;
11    int N = 10;
12    int quale;
13    Timer timer;
14    int width;
15    public Supercar() {
16      setBackground(Color.white);
17      setForeground(Color.pink);
18      timer = new Timer(WAIT, new Aggiorna());
19      timer.start();
20    }
21    public void paintComponent(Graphics g) {
22      super.paintComponent(g);
23      width = getWidth();
24      N = (width - (2 * BORDER)) / SPACE;
25      for (int i = 0; i < N; i++) {
26        if (i == quale)
27          g.setColor(Color.red);
28        g.fillRect(BORDER + (i * SPACE), BORDER, SIZE, SIZE);
29        g.setColor(Color.pink);
30      }
31    }
32    class Aggiorna implements ActionListener {
33      int dir = 1;
34      public void actionPerformed(ActionEvent ae) {
```

```
35          if (quale >= (N - 1))
36            dir = -1;
37          if (quale <= 0)
38            dir = 1;
39          quale += dir;
40          repaint();
41        }
42      }
43    }
```

Finestra.java:

```
1   package cap17.supercar;
2
3   import java.awt.*;
4   import javax.swing.*;
5   public class Finestra extends JFrame {
6     Supercar s;
7     public Finestra() {
8       super("Supercar");
9       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10      s = new Supercar();
11      getContentPane().add(s);
12      setSize(200, 100);
13    }
14    public static void main(String[] args) {
15      Finestra f = new Finestra();
16      f.setVisible(true);
17    }
18  }
```

Note e soluzione alternativa

La soluzione appena illustrata presenta due lievi difetti:

1. il calcolo del numero dei quadrati avviene ogni volta che il pannello viene ridisegnato, cioè ogni *WAIT* millisecondi (e non solo quando il pannello viene ridimensionato);
2. se non fosse per l'istruzione *setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)*, che istruisce la finestra a terminare tutti i thread della JVM quando viene premuto il bottone di chiusura, il thread associato al *Timer* impedirebbe all'applicazione di uscire.

Per risolvere il primo problema è sufficiente catturare gli eventi di ridimensionamento del pannello, in modo da ricalcolare solo in tali occasioni il nuovo valore di `N`. Questo richiede la presenza di una classe ascoltatore che implementa l'interfaccia *ComponentListener*: il metodo *void componentResized(ComponentEvent ce)* di tale interfaccia viene richiamato ogni volta che il componente a cui è associato l'ascoltatore viene ridimensionato.

Una classe ascoltatore che implementa direttamente *ComponentListener* deve implementare tutti i metodi dell'interfaccia e non solo *componentResized()*. Una soluzione più compatta può essere ottenuta attraverso una classe *adattatore*: il package *java.awt* contiene la definizione di un certo numero di classi che hanno un nome tipo *XxxAdapter*. La classe che ha nome *XxxAdapter* implementa l'interfaccia *XxxListener* fornendo una implementazione vuota dei metodi dell'interfaccia. In questo modo, se il programmatore ha la necessità di ridefinire solo uno dei metodi dell'interfaccia (spesso numerosi) può realizzare una classe ascoltatore che estende *XxxAdapter* ed in cui è presente la ridefinizione del solo metodo di suo interesse (ereditando l'implementazione fornita dall'adattatore di tutti gli altri metodi).

Per risolvere il secondo problema, la finestra deve catturare l'evento di pressione del bottone di chiusura e arrestare il thread associato al timer. Questo può essere ottenuto attraverso un ascoltatore che implementa l'interfaccia *WindowListener* (package *java.awt.event*) e associando l'ascoltatore alla finestra. Anche in questo caso, dovendo ridefinire il comportamento di uno solo dei metodi dell'interfaccia *WindowListener* è più conveniente l'uso di una classe adattatore (*WindowAdapter*).

Supercar2.java:

```
1  package cap17.supercar;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6  public class Supercar2 extends JPanel {
7    static final int SIZE = 10;
8    static final int SPACE = 12;
9    static final int BORDER = 5;
10   static final int WAIT = 300;
11   int N = 10;
12   int quale;
13   Timer timer;
14   int width;
15   public Supercar2() {
16     setBackground(Color.white);
17     setForeground(Color.pink);
18     addComponentListener(new AscoltatoreDimensioni());
```

```
19      timer = new Timer(WAIT, new Aggiorna());
20      timer.start();
21    }
22    public void paintComponent(Graphics g) {
23      super.paintComponent(g);
24      for (int i = 0; i < N; i++) {
25        if (i == quale)
26          g.setColor(Color.red);
27        g.fillRect(BORDER + (i * SPACE), BORDER, SIZE, SIZE);
28        g.setColor(Color.pink);
29      }
30    }
31    public void stop() {
32      timer.stop();
33    }
34    class Aggiorna implements ActionListener {
35      int dir = 1;
36      public void actionPerformed(ActionEvent ae) {
37        if (quale >= (N - 1))
38          dir = -1;
39        if (quale <= 0)
40          dir = 1;
41        quale += dir;
42        repaint();
43      }
44    }
45    class AscoltatoreDimensioni extends ComponentAdapter {
46      public void componentResized(ComponentEvent ce) {
47        width = getWidth();
48        N = (width - (2 * BORDER)) / SPACE;
49      }
50    }
51  }
```

Finestra2.java:

```
1   package cap17.supercar;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import javax.swing.*;
6   public class Finestra2 extends JFrame {
```

```
7     Supercar2 s;
8     public Finestra2() {
9       super("Supercar2");
10      setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
11      addWindowListener(new AscoltatoreFinestra());
12      s = new Supercar2();
13      getContentPane().add(s);
14      setSize(200, 100);
15    }
16    public static void main(String[] args) {
17      Finestra2 f2 = new Finestra2();
18      f2.setVisible(true);
19    }
20    class AscoltatoreFinestra extends WindowAdapter {
21      public void windowClosing(WindowEvent we) {
22        s.stop();
23      }
24    }
25  }
```

18. Applet e Servlet

18.1 Uomo cannone

Alfredo vuole diventare un uomo cannone ma non sa dove posizionare la rete per l'atterraggio. Per aiutare Alfredo a coronare il suo sogno realizzare un applet simile a quello mostrato in figura 18.1 in cui è possibile specificare l'angolo di sparo e la velocità iniziale. Quando viene premuto il bottone "Spara", viene visualizzata una animazione in cui un un pallino, accompagnato dalla scritta "Alfredo", si muove descrivendo il moto parabolico dell'uomo cannone.

Suggerimenti

La posizione in funzione del tempo è la seguente:
$x(t) = v_0 \cos\theta \cdot t$
$y(t) = v_0 \sin\theta \cdot t - \frac{1}{2}gt^2$
dove θ è l'angolo di sparo e v_0 la velocità iniziale.

Soluzione

UomoCannone.java:

```
1  package cap17.uomocan;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6  public class UomoCannone extends JApplet {
7    private static int DELAY = 50;
8    private JLabel angLab = new JLabel("Angolo (gradi)",
```

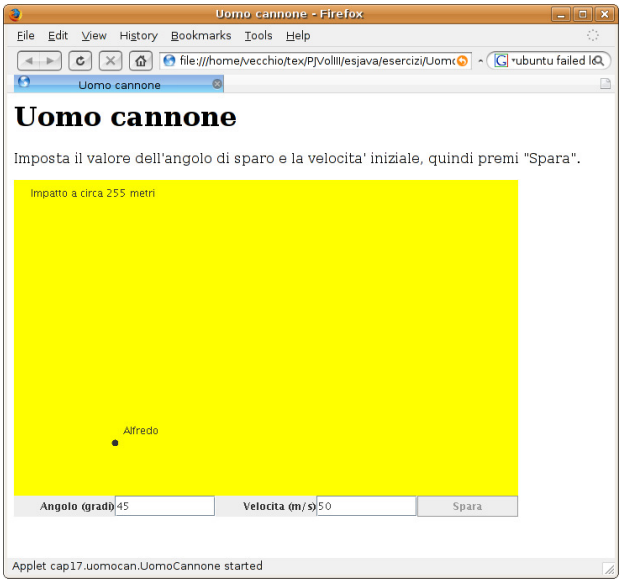


Figura 18.1: L'applet in una semplice pagina HTML

```
9                                       JLabel.TRAILING);
10    private JTextField angolo = new JTextField("45");
11    private JLabel velLab = new JLabel("Velocita (m/s)",
12                                       JLabel.TRAILING);
13    private JTextField velocita = new JTextField("50");
14    private JButton spara = new JButton("Spara");
15    private JPanel barra = new JPanel();
16    private Pannello p = new Pannello();
17    private Ascoltatore asc = new Ascoltatore();
18    private Timer timer;
19    public void init() {
20      barra.setLayout(new GridLayout(1, 5));
21      barra.add(angLab);
22      barra.add(angolo);
23      barra.add(velLab);
24      barra.add(velocita);
25      barra.add(spara);
26      spara.addActionListener(asc);
27      getContentPane().add(BorderLayout.PAGE_END, barra);
28      getContentPane().add(p);
```

```
29      }
30      public void stop() {
31        if (timer != null) {
32          timer.stop();
33        }
34      }
35      class Pannello extends JPanel {
36        private int x;
37        private int y;
38        private String messaggio;
39        private boolean primo = true;
40        Pannello() {
41          setBackground(Color.yellow);
42          messaggio = "";
43        }
44        void setMessaggio(String m) {
45          messaggio = m;
46        }
47        protected void paintComponent(Graphics g) {
48          super.paintComponent(g);
49          if (primo) {
50            y = getHeight();
51            primo = false;
52          }
53          g.drawString(messaggio, 20, 20);
54          g.fillOval(x - 4, y - 4, 8, 8);
55          g.drawString("Alfredo", x + 10, y - 10);
56        }
57        void aggiorna(double x, double y) {
58          this.x = (int) x;
59          this.y = getHeight() - (int) y;
60        }
61      }
62
63      class Aggiornatore implements ActionListener {
64        private final static double G = 9.8;
65        private double vx0;
66        private double vy0;
67        private long time = System.currentTimeMillis();
68
69        Aggiornatore(double a, double v) {
```

```
70          vx0 = v * Math.cos(a);
71          vy0 = v * Math.sin(a);
72          p.setMessaggio("Impatto a circa " + (int) ((
73                           2 * vx0 * vy0) / G) +
74                         " metri");
75        }
76
77        double getX() {
78          long t = System.currentTimeMillis() - time;
79          return (vx0 * t) / 1000.0;
80        }
81
82        double getY() {
83          double x = getX();
84          double y = ((vy0 * x) / vx0) - (0.5 * G * Math.pow(
85                                             x / vx0,
86                                             2));
87          return y;
88        }
89
90        public void actionPerformed(ActionEvent e) {
91          double x = getX();
92          double y = getY();
93          p.aggiorna(x, y);
94          p.repaint();
95          if (y < 0) {
96            timer.stop();
97            spara.setEnabled(true);
98          }
99        }
100     }
101
102     class Ascoltatore implements ActionListener {
103       public void actionPerformed(ActionEvent e) {
104         spara.setEnabled(false);
105         double angrad = Math.toRadians(Double.parseDouble(
106                                          angolo.getText()));
107         double vel = Double.parseDouble(velocita.getText());
108         timer = new Timer(DELAY, new Aggiornatore(angrad,
109                           vel));
110         timer.start();
```

```
111         }
112       }
113     }
```

pagina.html:

```
1   <html>
2   <title>Uomo cannone</title>
3   <body>
4   <h1>Uomo cannone</h1>
5   <p>
6   Imposta il valore dell'angolo di sparo e
7   la velocita' iniziale, quindi premi "Spara".
8   </p>
9   <applet code="cap17.uomocan.UomoCannone.class"
10   width=600 height=400> </applet>
11  </body>
12  </html>
```

18.2 Codice Fiscale

Realizzare un servlet che calcola il codice fiscale dell'utente. I dati vengono immessi attraverso un form simile a quello mostrato in Figura 18.2. Il codice fiscale viene calcolato secondo il seguente algoritmo (in realtà l'algoritmo per il calcolo del codice fiscale è più complesso di quello qui illustrato):

- il codice fiscale è composto da un numero pari ad undici di lettere e cifre;
- le prime tre lettere sono ricavate dalle consonanti del cognome; se il cognome contiene un numero di consonanti inferiore a tre, i caratteri mancanti sono posti uguali ad 'X';
- le successive tre lettere sono ricavate dalle consonanti del nome; se il nome contiene un numero di consonanti inferiore a tre, i caratteri mancanti sono posti uguali ad 'X';
- seguono le due cifre meno significative dell'anno di nascita;
- segue un carattere che corrisponde al mese di nascita ('A' per gennaio, 'B' per febbraio, e così via);
- seguono le due cifre del giorno di nascita (se l'utente è di sesso femminile, al giorno di nascita deve essere sommato il valore 40).

Per esempio, il codice fiscale di Mario Rossi, nato il 12/4/1955 è RSSMRX55D12.

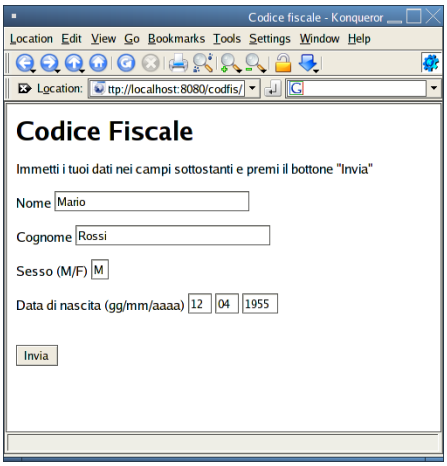


Figura 18.2: Form con cui l'utente inserisce i propri dati.

Soluzione

index.html:

```
1  <html>
2  <head>
3  <title>Codice fiscale</title>
4  </head>
5  <body>
6  <h1>Codice Fiscale</h1>
7  <p>
8  Immetti i tuoi dati nei campi sottostanti e premi il bottone "Invia"
9  </p>
10 <form name="input" action="codicefiscale">
11   <p>
12    Nome
13         <input type="text" name="nome" size="30">
14   </p>
15   <p>
16   Cognome
17         <input type="text" name="cognome" size="30">
18   </p>
19   <p>
20   Sesso (M/F)
```

```
21          <input type="text" name="sesso" size="1">
22    </p>
23    <p>
24    Data di nascita (gg/mm/aaaa)
25    <input type="text" name="giorno" size="2" maxlength="2">
26    <input type="text" name="mese" size="2" maxlength="2">
27    <input type="text" name="anno" size="4" maxlength="4">
28    </p>
29          <br>
30          <input type="submit" value="Invia">
31  </form>
32  </body>
33  </html>
```

web.xml:

```
1   <?xml version="1.0" encoding="ISO-8859-1"?>
2   <web-app>
3       <display-name>Codice Fiscale</display-name>
4       <description>
5         Calcola il codice fiscale dell'utente.
6       </description>
7       <servlet>
8         <servlet-name>codfis</servlet-name>
9         <description>
10          Servlet che calcola il codice fiscale dell'utente.
11        </description>
12        <servlet-class>cap18.codfis.CodiceFiscale</servlet-class>
13      </servlet>
14      <servlet-mapping>
15        <servlet-name>codfis</servlet-name>
16        <url-pattern>/codicefiscale</url-pattern>
17      </servlet-mapping>
18  </web-app>
```

CodiceFiscale.java:

```
1   package cap18.codfis;
2
3   import java.io.*;
4   import java.net.*;
5   import javax.servlet.*;
```

```
6   import javax.servlet.http.*;
7   public class CodiceFiscale extends HttpServlet {
8     static boolean consonante(char c) {
9       if ((c == 'A') || (c == 'E') || (c == 'I')
10          || (c == 'O')
11          || (c == 'U')) {
12        return false;
13      }
14      return true;
15    }
16    static String lettere(String s) {
17      int conta = 0;
18      char[] c = new char[3];
19      char[] d = s.toUpperCase().toCharArray();
20      for (int i = 0; (i < d.length) && (conta < 3); i++) {
21        if (consonante(d[i])) {
22          c[conta++] = d[i];
23        }
24      }
25      for (; conta < 3; conta++)
26        c[conta] = 'X';
27      return new String(c);
28    }
29    void processRequest(HttpServletRequest request,
30                        HttpServletResponse response)
31    throws ServletException, IOException {
32      response.setContentType("text/html");
33      PrintWriter out = response.getWriter();
34      out.println("<html>");
35      out.println("<head>");
36      out.println("<title>Codice Fiscale</title>");
37      out.println("</head>");
38      out.println("<body>");
39      out.println("<h1>Codice Fiscale</h1>");
40      String nome = request.getParameter("nome");
41      String cognome = request.getParameter("cognome");
42      String giorno = request.getParameter("giorno");
43      String mese = request.getParameter("mese");
44      String anno = request.getParameter("anno");
45      String sesso = request.getParameter("sesso");
46      String cn = lettere(nome);
```

```
47        String cc = lettere(cognome);
48        try {
49          String ca = anno.substring(anno.length() - 2);
50          int m = Integer.parseInt(mese);
51          char cm = (char) (('A' + m) - 1);
52          int g = Integer.parseInt(giorno);
53          if (sesso.toUpperCase().equals("F")) {
54            g += 40;
55          }
56          String codice = cc + cn + ca + cm + g;
57          out.println("<p>Il tuo codice fiscale e' " + codice);
58        } catch (NumberFormatException e) {
59          out.println("<p>Controlla la data di nascita</p>");
60        } catch (Exception e) {
61          out.println("<p>Controlla i tuoi dati</p>");
62        }
63        out.println("</body>");
64        out.println("</html>");
65        out.close();
66      }
67      protected void doGet(HttpServletRequest request,
68                           HttpServletResponse response)
69      throws ServletException, IOException {
70        processRequest(request, response);
71      }
72      protected void doPost(HttpServletRequest request,
73                            HttpServletResponse response)
74      throws ServletException, IOException {
75        processRequest(request, response);
76      }
77      public String getServletInfo() {
78        return "Servlet Codice Fiscale";
79      }
80    }
```

18.3 Indovina

Realizzare un servlet che chiede all'utente di indovinare una parola (Figura 18.3(a)) e, ad ogni tentativo, indica il numero di lettere indovinate (Figura 18.3(b)).

Suggerimenti

Usare il concetto di sessione per associare una parola da indovinare ad ogni cliente.

Soluzione

Indovina.java:

```
1  package cap18.indovina;
2
3  import java.io.*;
4  import java.net.*;
5  import javax.servlet.*;
6  import javax.servlet.http.*;
7  public class Indovina extends HttpServlet {
8    private static String[] parole = {
9      "oloturia", "visigoti", "bombolone", "rododendro",
10     "polpetta", "mescalina"
11   };
12   private static String parolaACaso() {
13     int n = (int) (Math.random() * parole.length);
14     return parole[n];
15   }
16   private static int quanteLettereCorrette(String a,
17       String b) {
18     a = a.toLowerCase();
19     b = b.toLowerCase();
20     int n = 0;
21     while ((a.length() > 0) && (b.length() > 0)) {
22       char c = a.charAt(0);
23       a = a.substring(1);
24       int index = b.indexOf(c);
25       if (index >= 0) {
26         n++;
27         if (index == 0)
28           b = b.substring(1);
29         else if (index == (b.length() - 1))
30           b = b.substring(0, index);
31         else
32           b = b.substring(0, index) +
33               b.substring(index + 1);
```

```
34          }
35        }
36        return n;
37      }
38      protected void doGet(
39        HttpServletRequest request,
40        HttpServletResponse response)
41      throws ServletException, IOException {
42        response.setContentType("text/html");
43        PrintWriter out = response.getWriter();
44        HttpSession s = request.getSession(true);
45        String str = parolaACaso();
46        s.setAttribute("segreto", str);
47        out.println("<html>");
48        out.println("<head>");
49        out.println("<title>Servlet Indovina</title>");
50        out.println("</head>");
51        out.println("<body>");
52        out.println("<h1>Indovina</h1>");
53        out.println(
54          "<p>Prova ad indovinare la parola a cui sto pensando ("
55          +
56          str.length() + " lettere)</p>");
57        out.println(
58          "<form name=\"input\" " + "action=\"indovina\" " +
59          "method=\"post\">");
60        out.println("<input type=\"text\" name=\"parola\">");
61        out.println(
62          "<input type=\"submit\" " + "value=\"Invia\">");
63        out.println("</body>");
64        out.println("</html>");
65        out.close();
66      }
67      protected void doPost(
68        HttpServletRequest request,
69        HttpServletResponse response)
70      throws ServletException, IOException {
71        response.setContentType("text/html");
72        PrintWriter out = response.getWriter();
73        out.println("<html>");
74        out.println("<head>");
```

```
75      out.println("<title>Servlet Indovina</title>");
76      out.println("</head>");
77      out.println("<body>");
78      out.println("<h1>Indovina</h1>");
79      HttpSession s = request.getSession(false);
80      if (s == null) {
81        out.println("<p>Errore: sessione non presente.</p>");
82        out.println("<p>Ritorna alla pagina iniziale.</p>");
83      } else {
84        String seg = (String) s.getAttribute("segreto");
85        String x = request.getParameter("parola");
86        boolean indovinato = x.equals(seg);
87        if (indovinato)
88          out.println("<p>Bravo! Hai indovinato!</p>");
89        else {
90          int q = quanteLettereCorrette(x, seg);
91          out.println(
92            "<p>" + q + " lettere indovinate, riprova</p>");
93          out.println(
94            "<form name=\"input\" " + "action=\"indovina\" " +
95            "method=\"post\">");
96          out.println(
97            "<input type=\"text\" name=\"parola\">");
98          out.println(
99            "<input type=\"submit\" " + "value=\"Invia\">");
100       }
101     }
102     out.println("</body>");
103     out.println("</html>");
104     out.close();
105   }
106   public String getServletInfo() {
107     return "Servlet Indovina";
108   }
109 }
```

web.xml:

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <web-app>
3      <display-name>Un semplice servlet</display-name>
4      <description>
```

```
5         Servlet che usa il concetto di sessione.
6       </description>
7       <servlet>
8         <servlet-name>indovina</servlet-name>
9         <description>
10          Chiede all'utente di indovinare una parola.
11        </description>
12        <servlet-class>cap18.indovina.Indovina</servlet-class>
13      </servlet>
14      <servlet-mapping>
15        <servlet-name>indovina</servlet-name>
16        <url-pattern>/indovina</url-pattern>
17      </servlet-mapping>
18  </web-app>
```

18.4 Sondaggio

Realizzare un servlet che chiede all'utente di partecipare ad un sondaggio, come mostrato in Figura 18.4(a). Quando l'utente esprime la sua scelta viene visualizzato il risultato del sondaggio (per esempio come illustrato in Figura 18.4(b)).

Soluzione

Vota.java:

```
1   package cap18.sondaggio;
2
3   import java.io.*;
4   import java.net.*;
5   import javax.servlet.*;
6   import javax.servlet.http.*;
7   public class Vota extends HttpServlet {
8     protected static String DOC_TYPE =
9       "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional//EN\" "
10      +
11      "\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\"";
12    protected String[] scelte = {
13      "Brasile", "Madagascar", "Birmania", "Messico",
14      "Sudafrica", "Mali", "Sri-lanka", "Tanzania"
15    };
16    protected int[] voti = new int[scelte.length];
```

```
17    protected void doGet(
18      HttpServletRequest request,
19      HttpServletResponse response)
20    throws ServletException, IOException {
21      response.setContentType("text/html");
22      PrintWriter out = response.getWriter();
23      out.println(DOC_TYPE);
24      out.println(
25        "<html><head><title>Vota</title></head><body>");
26      out.println("<p><b>Sondaggio:</b></p>");
27      out.println("<p>In quale paese vorresti andare ?</p>");
28      out.println(
29        "<form name=\"sondaggio\" action=\"\" method=\"post\">");
30      out.println("<table>");
31      for (int i = 0; i < scelte.length; i++) {
32        out.println("<tr><td>");
33        out.println(
34          "<input type=\"radio\" name=\"sondaggio\" value=\"" +
35          scelte[i] + "\" />");
36        out.println(scelte[i]);
37        out.println("</td></tr>");
38      }
39      out.println("</table>");
40      out.println("<input type=\"submit\" value=\"Vota\" />");
41      out.println("</form>");
42      out.println("</body></html>");
43      out.close();
44    }
45    protected void doPost(
46      HttpServletRequest request,
47      HttpServletResponse response)
48    throws ServletException, IOException {
49      response.setContentType("text/html");
50      PrintWriter out = response.getWriter();
51      out.println(DOC_TYPE);
52      out.println(
53        "<html><head><title>Risultati</title></head><body>");
54      String scelta = request.getParameter("sondaggio");
55      int j = 0;
56      for (; j < scelte.length; j++)
57        if (scelte[j].equals(scelta)) {
```

```
58            voti[j]++;
59            break;
60          }
61        if (j == scelte.length)
62          out.println("<p>Scelta non presente</p>");
63        else {
64          out.println("<p><b>Risultati:</b></p>");
65          out.println("<table>");
66          for (int i = 0; i < scelte.length; i++) {
67            out.println("<tr>");
68            out.println("<td>" + scelte[i] + "</td>");
69            out.println("<td>" + voti[i] + "</td>");
70            out.println("</tr>");
71          }
72          out.println("</table>");
73        }
74        out.println("</body></html>");
75        out.close();
76      }
77      public String getServletInfo() {
78        return "Servlet sondaggio";
79      }
80    }
```

web.xml:

```
1   <?xml version="1.0" encoding="ISO-8859-1"?>
2   <web-app>
3     <servlet>
4       <servlet-name>Vota</servlet-name>
5       <servlet-class>cap18.sondaggio.Vota</servlet-class>
6     </servlet>
7     <servlet-mapping>
8       <servlet-name>Vota</servlet-name>
9       <url-pattern>/vota</url-pattern>
10    </servlet-mapping>
11    <session-config>
12      <session-timeout>
13              30
14          </session-timeout>
15    </session-config>
16  </web-app>
```

18.5 Sondaggio corretto

Estendere l'esercizio precedente in modo tale che i votanti non possano esprimere una nuova preferenza se hanno già votato da meno di 24 ore.

Suggerimenti

Usare i cookie.

Soluzione

Vota2.java:

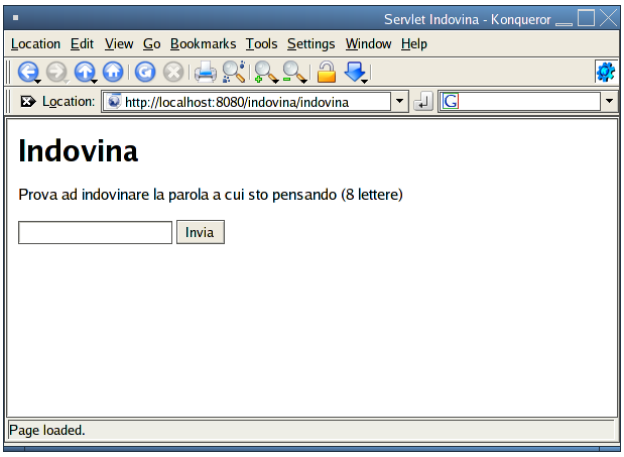
```
1  package cap18.sondcorr;
2
3  import cap18.sondaggio.Vota;
4  import java.io.*;
5  import java.net.*;
6  import java.util.*;
7  import javax.servlet.*;
8  import javax.servlet.http.*;
9  public class Vota2 extends Vota {
10   protected void doGet(
11     HttpServletRequest request,
12     HttpServletResponse response)
13   throws ServletException, IOException {
14     boolean votato = false;
15     Cookie[] cc = request.getCookies();
16     if (cc != null)
17       for (int i = 0; i < cc.length; i++)
18         if (cc[i].getName().equals("GiaVotato"))
19           votato = true;
20     if (!votato)
21       super.doGet(request, response);
22     else {
23       response.setContentType("text/html");
24       PrintWriter out = response.getWriter();
25       out.println(DOC_TYPE);
26       out.println(
27         "<html><head><title>Vota</title></head><body>");
28       out.println("<p><b>Hai gia' votato!</b></p>");
29       out.println("</body></html>");
```

```
30          out.close();
31        }
32      }
33      protected void doPost(
34        HttpServletRequest request,
35        HttpServletResponse response)
36      throws ServletException, IOException {
37        boolean votato = false;
38        Cookie[] cc = request.getCookies();
39        if (cc != null)
40          for (int i = 0; i < cc.length; i++)
41            if (cc[i].getName().equals("GiaVotato"))
42              votato = true;
43        if (!votato) {
44          Cookie c = new Cookie("GiaVotato", "");
45          c.setMaxAge(24 * 60 * 60);
46          response.addCookie(c);
47          super.doPost(request, response);
48        } else {
49          response.setContentType("text/html");
50          PrintWriter out = response.getWriter();
51          out.println(DOC_TYPE);
52          out.println(
53            "<html><head><title>Vota</title></head><body>");
54          out.println("<p><b>Hai gia' votato!</b></p>");
55          out.println("</body></html>");
56          out.close();
57        }
58      }
59      public String getServletInfo() {
60        return "Servlet sondaggio corretto";
61      }
62    }
```

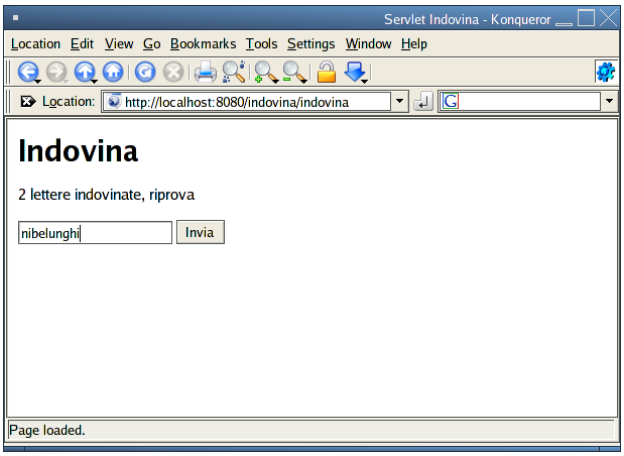
web.xml:

```
1   <?xml version="1.0" encoding="ISO-8859-1"?>
2   <web-app>
3     <servlet>
4       <servlet-name>Vota2</servlet-name>
5       <servlet-class>cap18.sondaggio.Vota2</servlet-class>
6     </servlet>
```

```
7     <servlet-mapping>
8       <servlet-name>Vota2</servlet-name>
9       <url-pattern>/vota2</url-pattern>
10    </servlet-mapping>
11    <session-config>
12      <session-timeout>
13              30
14      </session-timeout>
15    </session-config>
16  </web-app>
```

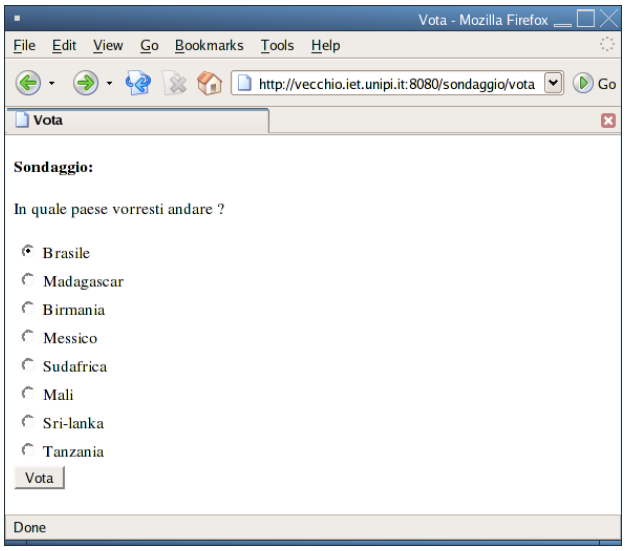


(a)

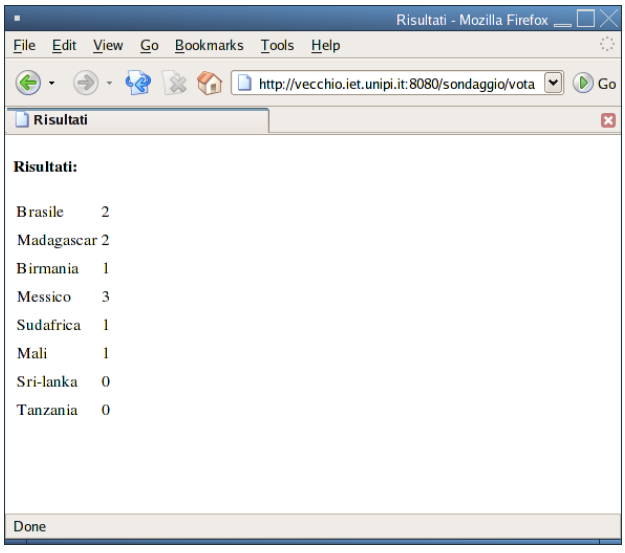


(b)

Figura 18.3: Pagine generate dal servlet.



(a)



(b)

Figura 18.4: Pagine generate dal servlet.

19. Strutture dati e Java collection framework

19.1 Tabella hash con liste

Nel Volume II è stata presentata la realizzazione di una tabella hash in cui i conflitti venivano risolti avanzando di un certo numero di posizioni all'interno della tabella stessa. Un modo diverso di risolvere i conflitti è quello di prevedere che ad ogni posizione i della tabella corrisponda una lista $l(i)$. La lista $l(i)$ conterrà tutte le coppie (k, v) (dove k è la chiave e v è il valore associato) tali che $hash(k) = i$.

Scrivere una class *TabellaHashList*, dotata degli stessi metodi della classe definita nel Volume II, ma realizzata nel modo descritto nel paragrafo precedente. Aggiungere, inoltre, i seguenti metodi:

- *int size()*: restituisce il numero di coppie memorizzate nella tabella.
- *void clear()*: svuota la tabella.
- *boolean isEmpty()*: restituisce *true* se la tabella è vuota, *false* altrimenti.

Soluzione

InsertException.java:

```
1  package cap19.tabellahash;
2
3  public class InsertException extends
4    RuntimeException {
5    public InsertException(String msg) {
```

```
6       super(msg);
7     }
8   }
```

TabellaHashList.java:

```
1   package cap19.tabellahash;
2
3   import java.util.*;
4   public class TabellaHashList {
5     static class Elem {
6       private Object key;
7       private Object info;
8       public Elem(Object k, Object o) {
9         key = k;
10        info = o;
11      }
12      public Object getKey() {
13        return key;
14      }
15      public Object getInfo() {
16        return info;
17      }
18      public String toString() {
19        return "k=" + key + ", o=" + info;
20      }
21    }
22    private LinkedList[] tab;
23    private int hash(Object k) {
24      if (k == null)
25        return 0;
26      return k.hashCode() % tab.length;
27    }
28    public TabellaHashList(int size) {
29      tab = new LinkedList[size];
30      for (int i = 0; i < size; i++)
31        tab[i] = new LinkedList();
32    }
33    public int size() {
34      int tot = 0;
35      for (int i = 0; i < tab.length; i++)
36        tot += tab[i].size();
```

```
37        return tot;
38      }
39      public void clear() {
40        for (int i = 0; i < tab.length; i++)
41          tab[i].clear();
42      }
43      public boolean isEmpty() {
44        for (int i = 0; i < tab.length; i++)
45          if (!tab[i].isEmpty())
46            return false;
47        return true;
48      }
49      public Object find(Object k) {
50        int h = hash(k);
51        Iterator i = tab[h].iterator();
52        while (i.hasNext()) {
53          Elem e = (Elem) i.next();
54          if (e.getKey().equals(k))
55            return e.getInfo();
56        }
57        return null;
58      }
59      public boolean insert(Object k, Object o) {
60        if (o == null)
61          throw new InsertException(
62            "Inserimento del valore null");
63        if (find(k) != null)
64          return false;
65        int h = hash(k);
66        tab[h].add(new Elem(k, o));
67        return true;
68      }
69      public Object extract(Object k) {
70        int h = hash(k);
71        Iterator i = tab[h].iterator();
72        while (i.hasNext()) {
73          Elem e = (Elem) i.next();
74          if (e.getKey().equals(k)) {
75            Object old = e.getInfo();
76            i.remove();
77            return old;
```

```
78          }
79        }
80        return null;
81      }
82      public String toString() {
83        String s = "";
84        for (int i = 0; i < tab.length; i++)
85          s += ("[" + i + "]" + tab[i] +
86                System.getProperty("line.separator"));
87        return s;
88      }
89    }
```

19.2 Coda prioritaria

Una coda prioritaria è una coda in cui, ad ogni elemento, è associata una priorità. L'operazione di estrazione restituisce sempre l'elemento inserito da più tempo tra quelli a maggiore priorità presenti nella coda. Si supponga che le priorità siano espresse da numeri naturali maggiori o uguali a zero, in modo che zero sia la priorità massima.

Utilizzando la classe *CodaSemplice* dell'esercizio 13.1, realizzare la classe *CodaPrioritaria* che implementi l'interfaccia *Coda<T>* (anch'essa definita nell'esercizio 13.1). Il metodo *out()* deve estrarre un elemento in accordo a quanto descritto nel paragrafo precedente. Il metodo *in()* deve inserire il nuovo elemento associandogli la priorità minima prevista. Inoltre, la classe *CodaPrioritaria* deve fornire i seguenti metodi.

- *CodaPrioritaria(int numPrio) throws ErrorePrio*: costruisce una nuova coda prioritaria con priorità che vanno da 0 a $numPrio - 1$. Solleva l'eccezione *ErrorePrio* se $numPrio \leq 0$.

- *void inPrio(T info, int prio) throws ErrorePrio*: inserisce in coda l'elemento *info* associandogli la priorità *prio*. Solleva l'eccezione *ErrorePrio* se *prio* non rientra nelle priorità consentite.

La classe *ErrorePrio* deve essere una eccezione contenente un messaggio del tipo "Priorità *x* non valida", dove il valore di *x* è passato all'oggetto eccezione al momento della creazione.

Suggerimenti

Un modo semplice di realizzare una coda prioritaria è quello di usare un array di code semplici, una per ogni priorità. Poiché, però, non è consentito, in Java, creare array di oggetti generici, si consiglia di usare la collezione ***ArrayList*<*Coda*<*T*>>**.

Soluzione

ErrorePrio.java:

```
1  package cap19.codaprioritaria;
2
3  public class ErrorePrio extends Exception {
4    ErrorePrio(int prio) {
5      super("Priorita' " + prio + " non valida");
6    }
7  }
```

CodaPrioritaria.java:

```
1  package cap19.codaprioritaria;
2
3  import cap13.coda.*;
4  import java.util.*;
5  public class CodaPrioritaria<T> implements Coda<T> {
6    private ArrayList<Coda<T>> code;
7    private int numPrio;
8    public CodaPrioritaria(int numPrio) throws
9      ErrorePrio {
10     if (numPrio <= 0)
11       throw new ErrorePrio(numPrio);
12     code = new ArrayList<Coda<T>>(numPrio);
13     for (int i = 0; i < numPrio; i++)
14       code.add(i, new CodaSemplice<T>());
15     this.numPrio = numPrio;
16   }
17   private Coda<T> primaNonVuota() {
18     for (Coda<T> c : code)
19       if (!c.vuota())
20         return c;
21     return null;
22   }
```

```
23      public boolean vuota() {
24        return primaNonVuota() == null;
25      }
26      public void in(T info) {
27        code.get(numPrio - 1).in(info);
28      }
29      public T out() throws CodaVuotaException {
30        Coda<T> c = primaNonVuota();
31        if (c == null)
32          throw new CodaVuotaException();
33        return c.out();
34      }
35      public void inPrio(T info,
36                         int prio) throws ErrorePrio {
37        if ((prio < 0) || (prio >= numPrio))
38          throw new ErrorePrio(numPrio);
39        code.get(prio).in(info);
40      }
41    }
```

19.3 Grafo

Un *grafo* (orientato) è una coppia (V,E) dove V è un insieme di nodi, e $E \subseteq V \times V$ è un insieme di archi. Dato un arco $a = (n,m)$, si dice che n è il nodo sorgente di a, che m è il nodo destinazione di a, che a è un arco uscente da n e che m è adiacente a n.

Scrivere le seguenti classi che modellano un grafo. Dove viene usata la parola "uguale", si intende sempre l'uguaglianza tramite il metodo *equals()*.

La classe *Arco* modella un arco, con costruttore *Arco(Nodo s, Nodo d)* e metodi (di lettura) *Nodo sorgente()* e *Nodo destinazione()*.

La classe *Nodo* modella un nodo del grafo. Il nodo memorizza anche l'insieme degli archi uscenti dal nodo. Prevede un costruttore senza argomenti e i metodi seguenti.

- *boolean aggiungiArco(Arco a)*: aggiunge un arco all'insieme degli archi uscenti dal nodo. Restituisce *false* se esiste già un arco uguale ad a (nel qual caso a non viene aggiunto), *true* altrimenti.

- *boolean rimuoviArco(Arco a)*: rimuove l'arco (se esiste) uguale ad a dall'insieme degli archi uscenti. Restituisce *true* se tale arco esiste, e *false* altrimenti.

- *void rimuoviAdiacenza(Nodo n)*: rimuove tutti gli archi uscenti che hanno un nodo destinazione uguale a *n*.
- *Iterator<Arco> archiUscenti()*: restituisce un iteratore che permetta di scorrere tutti gli archi uscenti.
- *Iterator<Nodo> nodiAdiacenti()*: restituisce un iteratore che permetta di scorrere tutti i nodi uscenti.

La classe *Grafo* modella un grafo. Prevede un costruttore senza argomenti e i seguenti metodi.

- *boolean aggiungiNodo(Nodo n)*: aggiunge *n* all'insieme dei nodi del grafo. Restituisce *false* se esiste già un nodo uguale a *n* (nel qual caso *n* non viene aggiunto), *false* altrimenti.
- *boolean rimuoviNodo(Nodo n)*: rimuove il nodo (se esiste) uguale ad *n*. Restituisce *true* se tale nodo esiste, e *false* altrimenti. Oltre al nodo, devono essere rimossi dal grafo anche tutti gli archi che hanno il nodo in questione come sorgente o destinazione.
- *boolean aggiungiArco(Arco a)*: aggiunge l'arco *a* al grafo. Aggiunge anche i nodi sorgenti e destinazione di *a* (se non ve ne erano già di uguali). Restituisce *false* (e non aggiunge l'arco) se il grafo conteneva già un arco uguale ad *a*, *true* altrimenti.
- *boolean rimuoviArco(Arco a)*: rimuove l'arco (se esiste) uguale ad *a*. Restituisce *true* se tale arco esiste, e *false* altrimenti.
- *Iterator<Arco> dammiArchi()*: restituisce un iteratore che permetta di scorrere tutti gli archi del grafo.
- *Iterator<Nodo> dammiNodi()*: restituisce un iteratore che permetta di scorrere tutti i nodi del grafo.

Soluzione

Arco.java:

```
1  package cap19.grafo;
2
3  public class Arco {
4    private Nodo s;
5    private Nodo d;
```

```
6
7     public Arco(Nodo s, Nodo d) {
8       this.s = s;
9       this.d = d;
10    }
11
12    public Nodo sorgente() {
13      return s;
14    }
15
16    public Nodo destinazione() {
17      return d;
18    }
19  }
```

Nodo.java:

```
1   package cap19.grafo;
2
3   import cap13.insieme.*;
4   import java.util.Iterator;
5   public class Nodo {
6     private Insieme<Arco> archi;
7     public Nodo() {
8       archi = new InsiemeList<Arco>();
9     }
10    public boolean aggiungiArco(Arco a) {
11      return archi.aggiungi(a);
12    }
13    public boolean rimuoviArco(Arco a) {
14      return archi.rimuovi(a);
15    }
16    public void rimuoviAdiacenza(Nodo n) {
17      Iterator<Nodo> i = nodiAdiacenti();
18      while (i.hasNext()) {
19        Nodo m = i.next();
20        if (m.equals(n))
21          i.remove();
22      }
23    }
24    public Iterator<Arco> archiUscenti() {
25      return archi.iterator();
```

```
26      }
27      public Iterator<Nodo> nodiAdiacenti() {
28        return new IteratoreNodi(archi.iterator());
29      }
30      private class IteratoreNodi implements
31        Iterator<Nodo> {
32        private Iterator<Arco> i;
33        IteratoreNodi(Iterator<Arco> i) {
34          this.i = i;
35        }
36        public Nodo next() throws IllegalStateException {
37          return i.next().destinazione();
38        }
39        public boolean hasNext() {
40          return i.hasNext();
41        }
42        public void remove() throws IllegalStateException {
43          i.remove();
44        }
45      }
46    }
```

Grafo.java:

```
1   package cap19.grafo;
2
3   import cap13.insieme.*;
4   import java.util.Iterator;
5   public class Grafo {
6     private Insieme<Nodo> nodi;
7     private Nodo cerca(Nodo n) {
8       Iterator<Nodo> i = dammiNodi();
9       while (i.hasNext()) {
10        Nodo m = i.next();
11        if (m.equals(n))
12          return m;
13      }
14      return null;
15    }
16    public Grafo() {
17      nodi = new InsiemeList<Nodo>();
18    }
```

```
19    public boolean aggiungiNodo(Nodo n) {
20      return nodi.aggiungi(n);
21    }
22    public boolean rimuoviNodo(Nodo n) {
23      if (nodi.rimuovi(n)) {
24        Iterator<Nodo> i = dammiNodi();
25        while (i.hasNext())
26          i.next().rimuoviAdiacenza(n);
27        return true;
28      }
29      return false;
30    }
31    public boolean aggiungiArco(Arco a) {
32      aggiungiNodo(a.sorgente());
33      aggiungiNodo(a.destinazione());
34      return cerca(a.sorgente()).aggiungiArco(a);
35    }
36    public boolean rimuoviArco(Arco a) {
37      Nodo n = cerca(a.sorgente());
38      if (n == null)
39        return false;
40      else
41        return n.rimuoviArco(a);
42    }
43    public Iterator<Nodo> dammiNodi() {
44      return nodi.iterator();
45    }
46    public Iterator<Arco> dammiArchi() {
47      return new IteratoreArchi(dammiNodi());
48    }
49    private class IteratoreArchi implements
50      Iterator<Arco> {
51      Iterator<Nodo> i;
52      Iterator<Arco> j;
53      IteratoreArchi(Iterator<Nodo> i) {
54        this.i = i;
55        vaiPrimoConArchi();
56      }
57      private void vaiPrimoConArchi() {
58        j = null;
59        while (i.hasNext()) {
```

```
60            j = i.next().archiUscenti();
61            if (j.hasNext())
62              return;
63            j = null;
64          }
65        }
66        public boolean hasNext() {
67          if (j == null)
68            return false;
69          if (j.hasNext())
70            return true;
71          vaiPrimoConArchi();
72          return j != null;
73        }
74        public Arco next() throws IllegalStateException {
75          if (j == null)
76            throw new IllegalStateException();
77          if (j.hasNext())
78            return j.next();
79          vaiPrimoConArchi();
80          if (j == null)
81            throw new IllegalStateException();
82          return j.next();
83        }
84        public void remove() throws IllegalStateException {
85          if (j == null)
86            throw new IllegalStateException();
87          j.remove();
88        }
89      }
90    }
```

19.4 Grafo connesso

Scrivere una classe *AlgoGrafi* contenente i seguenti metodi statici, che operano su oggetti di tipo *Nodo* e *Grafo*, come definiti nell'esercizio 19.3.

- *Insieme<Nodo> raggiungibili(Nodo n)*: restituisce l'insieme dei nodi raggiungibili partendo dal nodo *n*. Un nodo *m* è raggiungibile da un nodo *n* se esiste un cammino orientato che parte da *n* e arriva in *m*.

- *boolean connesso(Grafo g)*: restituisce *true* se il grafo g è connesso. Un grafo è detto *connesso* se ogni coppia di nodi è collegata da un cammino non orientato.

Suggerimenti

Per realizzare la seconda funzione, può essere utile costruire prima un grafo non orientato a partire dal grafo g. Per fare ciò, si deve costruire un nuovo grafo *g1* che contenga un nodo n' per ogni nodo n contenuto in g. Quindi, per ogni arco (n,m) contenuto in g, il grafo *g1* dovrà contenere gli archi (n',m') e (m',n').

Soluzione

AlgoGrafi.java:

```
1  package cap19.grafoconnesso;
2
3  import cap13.insieme.*;
4  import cap19.grafo.*;
5  import cap19.tabellahash.*;
6  import java.util.Iterator;
7  public class AlgoGrafi {
8    private static void visita(
9      Nodo n, Insieme<Nodo> visitati) {
10     Iterator<Nodo> i = n.nodiAdiacenti();
11     while (i.hasNext()) {
12       Nodo m = i.next();
13       if (visitati.contiene(m))
14         continue;
15       visitati.aggiungi(m);
16       visita(m, visitati);
17     }
18   }
19   public static Insieme<Nodo> raggiungibili(Nodo n) {
20     Insieme<Nodo> visitati = new InsiemeList<Nodo>();
21     visitati.aggiungi(n);
22     visita(n, visitati);
23     return visitati;
24   }
25   private static Grafo bidir(Grafo g) {
26     TabellaHashList t = new TabellaHashList(10);
27     Iterator<Nodo> i = g.dammiNodi();
```

```
28        Grafo g1 = new Grafo();
29        while (i.hasNext()) {
30          Nodo n = i.next();
31          Nodo m = new Nodo();
32          t.insert(n, m);
33          g1.aggiungiNodo(m);
34        }
35        Iterator<Arco> j = g.dammiArchi();
36        while (j.hasNext()) {
37          Arco a = j.next();
38          Nodo s = (Nodo) t.find(a.sorgente());
39          Nodo d = (Nodo) t.find(a.destinazione());
40          g1.aggiungiArco(new Arco(s, d));
41          g1.aggiungiArco(new Arco(d, s));
42        }
43        return g1;
44      }
45      public static boolean connesso(Grafo g) {
46        Grafo g1 = bidir(g);
47        Iterator<Nodo> i = g1.dammiNodi();
48        while (i.hasNext()) {
49          Nodo n = i.next();
50          Insieme<Nodo> visitati = raggiungibili(n);
51          Iterator<Nodo> j = g1.dammiNodi();
52          while (j.hasNext()) {
53            Nodo m = j.next();
54            if (!visitati.contiene(m))
55              return false;
56          }
57        }
58        return true;
59      }
60    }
```

19.5 Albero iterabile

Si scriva una classe *Albero*, che realizza un albero binario. Ogni nodo dell'albero memorizza una informazione di tipo *Object*. La classe deve fornire i seguenti tipi e metodi:

- *Order*: un tipo enumerato con valori *PRE* e *POST*.

- *Albero(Object o)*: costruttore che inizializza un nuovo albero composto da un unico nodo. Il nodo memorizzerà l'informazione *o*.
- *Object getInfo()*: restituisce l'informazione associata alla radice dell'albero.
- *void setDes(Albero a)* e *void setSin(Albero a)*: l'albero *a*, passato per argomento, deve diventare il sottoalbero destro (risp. sinistro) dell'albero a cui le funzioni sono applicate.
- *Albero getDes()* e *Albero getSin()*: restituiscono il sottoalbero destro (risp. sinistro) della radice dell'albero a cui le funzioni sono applicate.
- *Iterator getIterator(int type)*: restituisce un *java.util.Iterator* che permette di scorrere tutti i nodi dell'albero. Il parametro *type* può assumere i valori *Albero.Order.PRE* oppure *Albero.Order.POST*. Nel primo caso, l'iteratore permetterà di scorrere i nodi dell'albero in ordine anticipato e in ordine posticipato nel secondo. Non è richiesto il metodo *remove()* (è possibile lanciare l'eccezione *OperationNotSupportedExceptio*).

Suggerimenti

Si ricorda che l'ordine anticipato prevede che ogni nodo venga visitato prima dei suoi figli, mentre l'ordine posticipato, al contrario, prevede che ogni nodo venga visitato dopo i suoi figli.

Per realizzare l'iteratore (anticipato o posticipato) di tutto l'albero, si consiglia di realizzare, prima, un iteratore di singolo nodo (che permetta di scorrere i due figli del nodo associato).

Si noti, infine che, poiché la visita deve avvenire un passo alla volta (dove ogni passo coincide con una invocazione del metodo *next()*), non è possibile utilizzare la classica realizzazione ricorsiva. Ogni iteratore, quindi, dovrà ricordare (ad esempio usando una pila) i nodi che devono essere ancora visitati...

Soluzione

Albero.java:

```
1  package cap19.alberoiterabile;
2
3  import IngressoUscita.Console;
4  import cap9.pila.*;
5  import java.util.Iterator;
6  import java.util.NoSuchElementException;
7  public class Albero {
```

```
8     enum Order {
9       PRE, POST;
10    }
11    private Object info;
12    private Albero sin;
13    private Albero des;
14    public Albero(Object o) {
15      info = o;
16    }
17    public void setSin(Albero a) {
18      sin = a;
19    }
20    public void setDes(Albero a) {
21      des = a;
22    }
23    public Albero getSin() {
24      return sin;
25    }
26    public Albero getDes() {
27      return des;
28    }
29    public Object getInfo() {
30      return info;
31    }
32    public Iterator getIterator(Order o) {
33      switch (o) {
34      case POST:
35        return new IteratorePosticipato();
36      case PRE:
37        return new IteratoreAnticipato();
38      default:
39        return null;
40      }
41    }
42    private class IteratoreNodo implements Iterator {
43      boolean n;
44      boolean s;
45      boolean d;
46      private IteratoreNodo() {
47        s = sin != null;
48        d = des != null;
```

```
49      }
50      public Object next() {
51        if (s) {
52          s = false;
53          return sin;
54        } else if (d) {
55          d = false;
56          return des;
57        } else
58          throw new NoSuchElementException();
59      }
60      public boolean hasNext() {
61        return s || d;
62      }
63      public void remove() {
64        throw new UnsupportedOperationException();
65      }
66    }
67    private class IteratoreAnticipato implements
68      Iterator {
69      protected Pila p = new Pila();
70      private IteratoreAnticipato() {
71        Albero dummy = new Albero(null);
72        dummy.setSin(Albero.this);
73        p.push(dummy.new IteratoreNodo());
74      }
75      public boolean hasNext() {
76        return !p.isEmpty();
77      }
78      public Object next() {
79        if (p.isEmpty())
80          throw new NoSuchElementException();
81        IteratoreNodo t = (IteratoreNodo) p.top();
82        Albero a = (Albero) t.next();
83        p.push(a.new IteratoreNodo());
84        while (
85          !p.isEmpty() &&
86          !((IteratoreNodo) p.top()).hasNext())
87          p.pop();
88        return a;
89      }
```

```
90        public void remove() {
91          throw new UnsupportedOperationException();
92        }
93      }
94      private class IteratorePosticipato implements
95        Iterator {
96        protected Pila pa = new Pila();
97        protected Pila pi = new Pila();
98        public IteratorePosticipato() {
99          Albero dummy = new Albero(null);
100         dummy.setSin(Albero.this);
101         goDown(dummy.new IteratoreNodo());
102       }
103       public Object next() {
104         if (pa.isEmpty())
105           throw new NoSuchElementException();
106         Albero a = (Albero) pa.pop();
107         IteratoreNodo i = (IteratoreNodo) pi.pop();
108         if (i.hasNext())
109           goDown(i);
110         return a;
111       }
112       public boolean hasNext() {
113         return !pa.isEmpty();
114       }
115       public void remove() {
116         throw new UnsupportedOperationException();
117       }
118       private void goDown(IteratoreNodo i) {
119         while (i.hasNext()) {
120           Albero a = (Albero) i.next();
121           pa.push(a);
122           pi.push(i);
123           i = a.new IteratoreNodo();
124         }
125       }
126     }
127   }
```

Note

Nelle righe 44–70 viene realizzato l'iteratore di un singolo nodo: l'iteratore mantiene una coppia di variabili booleane, che usa per ricordare quali dei due figli del nodo associato (se esistono) deve essere ancora visitato.

La classe *IteratoreAnticipato* realizza la visita in ordine anticipato. L'idea è di usare una *Pila* di *IteratoreNodo*. In cima alla pila c'è sempre l'iteratore associato al nodo *n* di cui si stanno visitando i figli. Nelle altre posizioni della pila ci sono gli iteratori associati ai nodi che si trovano sul percorso dalla radice dell'albero fino a *n*. Ogni iteratore in pila ricorda quali figli del nodo associato devono essere ancora visitati. Il metodo *next()* (linee 81–91) funziona nel seguente modo. Il nodo *n* da restituire in questa istanza è il prossimo nodo dell'iteratore in cima alla pila (linee 84–85). La prossima volta che verrà chiamato il metodo *next()*, bisognerà visitare i figli di *n*, quindi l'iteratore del nodo *n* viene posto in cima alla pila (linea 86). Le linee 87–90 hanno lo scopo di eliminare dalla pila tutti gli iteratori terminati, in modo che la prossima invocazione del metodo *next()* possa limitarsi a controllare che la pila non sia vuota (linee 82–83) ed estrarre il primo iteratore dalla pila (linea 84), avendo la certezza che tale iteratore non è terminato (linea 85). Si noti che, poiché, con questa realizzazione, si visitano solo nodi che sono figli di qualche altro nodo, è necessario introdurre un nodo fittizio che faccia da padre al nodo radice dell'albero che, altrimenti, non verrebbe visitato (righe 73–75).

La classe *IteratorePosticipato* ha bisogno di due pile, *pa* e *pi* (linee 97–98) utilizzate nel modo seguente: ad ogni istante, la cima della pila *pa* contiene il prossimo nodo da visitare, mentre la pila *pi* contiene l'iteratore che punta ai fratelli di questo nodo (se ve sono). Il costruttore (linee 100–102) utilizza la stessa tecnica dell'albero fittizio, vista per *IteratoreAnticipato*. In più, poiché i nodi figli devono essere visitati prima dei nodi genitori, le due pile devono essere precaricate, in modo da portarsi nella situazione in cui il prossimo nodo da visitare sia il nodo senza figli più a sinistra. Tale precaricamento viene svolto dalla funzione *goDown()* (linee 119–126). Il metodo *next()* funziona nel modo illustrato di seguito. Il nodo *n* da restituire e l'iteratore di nodo corrente (che punta ai fratelli del nodo *n*) vengono estratti dalle rispettive pile (linee 107 e 108). Se il nodo *n* ha fratelli (linea 109), allora le pile vengono precaricate, in modo che il prossimo nodo da restituire sia il nodo senza figli più a sinistra, discendente del primo fratello del nodo *n* (linea 110).

19.6 Confronto prestazioni

Ci proponiamo di confrontare le prestazioni di due o più realizzazioni della stessa interfaccia. Consideriamo per esempio l'interfaccia *List*, di cui il Java Collection Framework contiene due implementazioni distinte: *ArrayList* e *LinkedList*. Vogliamo sapere quale delle due implementazioni è più veloce nello svolgere operazioni

come inserimento in testa, inserimento in una posizione qualunque, accesso ad un elemento qualunque, estrazione, etc.

Definiamo a questo scopo la seguente classe:

Statistica.java:

```
1  package cap19.prestazioni;
2
3  public class Statistica {
4    private String nome;
5    private double tempo;
6    public Statistica(String nome) {
7      this.nome = nome;
8    }
9    public String dammiNome() {
10     return nome;
11   }
12   public void setTempo(double tempo) {
13     this.tempo = tempo;
14   }
15   public double dammiTempo() {
16     return tempo;
17   }
18   public String toString() {
19     return nome + ": " + tempo;
20   }
21 }
```

L'idea è che ogni statistica sia rappresentata da un testo descrittivo (per esempio, "Inserimento in testa") e un numero (per esempio, tempo medio necessario per inserire un elemento in testa alla lista). Definiamo inoltre la seguente interfaccia:

Profile.java:

```
1  package cap19.prestazioni;
2
3  public interface Profile {
4    int tipi();
5
6    public Statistica[] start(int tipo, long seme);
7  }
```

Una classe *P* che implementi l'interfaccia *Profile* rappresenta una serie di misure di prestazioni eseguite su una data interfaccia, di cui esistono uno o più *tipi* di

implementazioni diverse. Il metodo *tipi()* deve restituire il numero di tipi di implementazioni della stessa interfaccia note all'oggetto di classe *P*. Se tale numero è n, supponiamo che i diversi tipi siano identificati da un numero da 0 a $n-1$. Il metodo *start(int tipo, long seme)* deve eseguire tutte le misure previste, sull'implementazione numero *tipo*, e restituire i risultati sotto forma di un array di *Statistica*. Il parametro *seme* deve servire ad inizializzare il generatore di numeri casuali, che, tipicamente, viene usato quando si eseguono delle misure di prestazioni.

Realizzare una classe *ProfileList*, con lo scopo di misurare le prestazioni di diverse implementazioni dell'interfaccia *List*. La classe deve prevedere un costruttore *ProfileList(List[] liste)*, dove *liste* è un array contenente tutti gli oggetti di tipo *List* su cui eseguire le statistiche (l'elemento in posizione *i*-esima all'interno di questo array rappresenta l'*i*-esimo tipo di implementazione). Il metodo *start()* deve calcolare il tempo medio (per elemento) necessario ad eseguire ognuna delle seguenti operazioni (ogni operazione corrisponde ad un oggetto *Statistica* da restituire). Supponiamo che N sia una opportuna costante, che le liste contengano elementi di tipo *int* e che ogni operazione successiva lavori sulla lista così come lasciata dall'operazione precedente.

1. Inserimento in testa di $N/10$ elementi, con informazioni $i = 0, \ldots N/10-1$.
2. Inserimento di N elementi, con informazioni $i = 0, \ldots N-1$, in una posizione j scelta casualmente (ma esistente) per ogni elemento da inserire.
3. Ricerca di $N/4$ elementi contenenti una informazione i scelta a caso tra 0 e N.
4. Rimozione dalla testa di $N/10$ elementi.
5. Rimozione di $N/4$ elementi da una posizione casuale (esistente).
6. Rimuovere $N/4$ elementi da una posizione (esistente) scelta a caso, ma tramite un iteratore: per ogni elemento, ottenere un nuovo iteratore, farlo avanzare fino alla posizione scelta e quindi usare il metodo *remove()* dell'iteratore.

Infine, scrivere una classe *Test* che allochi un oggetto *ProfileList* passandogli una lista di tipo *ArrayList* e una di tipo *LinkedList* (entrambe inizialmente vuote). Quindi, invochi il metodo *start()* prima per il tipo 0 (*ArrayList*) e poi per il tipo 1 (*LinkedList*), stampando i risultati di tutte le statistiche così ottenute. Per eseguire un confronto equo, passare lo stesso valore di *seme* in entrambi i casi.

Soluzione

ProfileList.java:

```
1  package cap19.prestazioni;
2
3  import java.util.*;
4  public class ProfileList implements Profile {
5    private static final int N = 100000;
6    List[] liste;
7    doStat[] tests;
8    Random r;
9    public ProfileList(List[] liste) {
10     this.liste = liste;
11     tests = new doStat[6];
12     tests[0] = new InsTesta();
13     tests[1] = new InsCasuale();
14     tests[2] = new RicercaCasuale();
15     tests[3] = new RimTesta();
16     tests[4] = new RimCasuale();
17     tests[5] = new RimIterator();
18   }
19   public int tipi() {
20     return liste.length;
21   }
22   public Statistica[] start(int tipo, long seme) {
23     Statistica[] stats = new Statistica[tests.length];
24     r = new Random();
25     long start;
26     long stop;
27     r.setSeed(seme);
28     for (int i = 0; i < tests.length; i++) {
29       stats[i] = tests[i].apri(tipo);
30       start = System.currentTimeMillis();
31       tests[i].fai();
32       stop = System.currentTimeMillis();
33       tests[i].chiudi(stats[i], stop - start);
34     }
35     return stats;
36   }
37   private interface doStat {
38     Statistica apri(int tipo);
39     void fai();
40     void chiudi(Statistica s, long tempoTrascorso);
41   }
```

```
42    private class InsTesta implements doStat {
43      List l;
44
45      public Statistica apri(int tipo) {
46        l = liste[tipo];
47        return new Statistica("Iserimento in testa");
48      }
49
50      public void fai() {
51        for (int i = 0; i < (N / 10); i++)
52          l.add(i);
53      }
54
55      public void chiudi(Statistica s,
56                         long tempoTrascorso) {
57        s.setTempo((double) tempoTrascorso / (N / 10));
58      }
59    }
60
61    private class InsCasuale implements doStat {
62      List l;
63
64      public Statistica apri(int tipo) {
65        l = liste[tipo];
66        return new Statistica("Iserimento casuale");
67      }
68
69      public void fai() {
70        for (int i = 0; i < N; i++) {
71          int j = r.nextInt(l.size() + 1);
72          l.add(j, i);
73        }
74      }
75
76      public void chiudi(Statistica s,
77                         long tempoTrascorso) {
78        s.setTempo((double) tempoTrascorso / N);
79      }
80    }
81
82    private class RicercaCasuale implements doStat {
```

```
83        List l;
84        int size;
85
86        public Statistica apri(int tipo) {
87          l = liste[tipo];
88          size = l.size();
89          return new Statistica("Ricerca casuale");
90        }
91
92        public void fai() {
93          for (int i = 0; i < (N / 4); i++) {
94            int j = r.nextInt(size);
95            l.get(j);
96          }
97        }
98
99        public void chiudi(Statistica s,
100                           long tempoTrascorso) {
101         s.setTempo((double) tempoTrascorso / (N / 4));
102       }
103     }
104
105     private class RimTesta implements doStat {
106       List l;
107
108       public Statistica apri(int tipo) {
109         l = liste[tipo];
110         return new Statistica("Rimozione Testa");
111       }
112
113       public void fai() {
114         for (int i = 0; i < (N / 10); i++)
115           l.remove(0);
116       }
117
118       public void chiudi(Statistica s,
119                          long tempoTrascorso) {
120         s.setTempo((double) tempoTrascorso / (N / 10));
121       }
122     }
123
```

```
124     private class RimCasuale implements doStat {
125       List l;
126
127       public Statistica apri(int tipo) {
128         l = liste[tipo];
129         return new Statistica("Rimozione Casuale");
130       }
131
132       public void fai() {
133         for (int i = 0; i < (N / 4); i++) {
134           int j = r.nextInt(l.size());
135           l.remove(j);
136         }
137       }
138
139       public void chiudi(Statistica s,
140                          long tempoTrascorso) {
141         s.setTempo((double) tempoTrascorso / (N / 4));
142       }
143     }
144
145     private class RimIterator implements doStat {
146       List l;
147
148       public Statistica apri(int tipo) {
149         l = liste[tipo];
150         return new Statistica("Rimozione tramite Iterator");
151       }
152
153       public void fai() {
154         for (int i = 0; i < (N / 4); i++) {
155           int j = r.nextInt(l.size() - 1) + 1;
156           Iterator k = l.iterator();
157           for (int n = 0; n < j; n++)
158             k.next();
159           k.remove();
160         }
161       }
162
163       public void chiudi(Statistica s,
164                          long tempoTrascorso) {
```

```
165         s.setTempo((double) tempoTrascorso / (N / 4));
166       }
167     }
168   }
```

Test.java:

```
1   package cap19.prestazioni;
2
3   import IngressoUscita.Console;
4   import java.util.*;
5   class Test {
6     public static void main(String[] args) {
7       List[] l = new List[2];
8       Statistica[] stats;
9       l[0] = new ArrayList();
10      l[1] = new LinkedList();
11      Profile p = new ProfileList(l);
12      long seed = System.currentTimeMillis();
13      stats = p.start(0, seed);
14      Console.scriviStringa("ArrayList:");
15      for (Statistica s : stats)
16        Console.scriviStringa(s.toString());
17      stats = p.start(1, seed);
18      Console.scriviStringa("LinkedList:");
19      for (Statistica s : stats)
20        Console.scriviStringa(s.toString());
21    }
22  }
```

Note

Se proviamo ad eseguire il programma, otteniamo un output del genere (i valori precisi dipendono dalla velocità del processore e dalle condizioni di carico del sistema durante il test):

```
ArrayList:
Iserimento in testa: 0.0017
Iserimento casuale: 0.10112
Ricerca casuale: 5.6E-4
Rimozione Testa: 0.3096
Rimozione Casuale: 0.12984
```

```
Rimozione tramite Iterator: 2.34976
LinkedList:
Iserimento in testa: 6.0E-4
Iserimento casuale: 0.65312
Ricerca casuale: 1.72248
Rimozione Testa: 6.0E-4
Rimozione Casuale: 1.3946
Rimozione tramite Iterator: 2.71748
```

Notiamo come la classe *LinkedList* sia sensibilmente più veloce della classe *ArrayList* solo per le operazioni di inserimento in testa e rimozione dalla testa. Ciò è dovuto al fatto che, per eseguire tali operazioni, la classe *ArrayList* deve ricopiare tutti gli elementi contenuti nella lista. Notiamo, invece, come la classe *ArrayList* sia molto più veloce della classe *LinkedList* non solo per le operazioni di ricerca casuale (come è facile prevedere), ma anche per le operazioni di inserimento casuale e rimozione casuale. Ciò è dovuto al fatto che la classe *LinkedList* deve percorrere tutta la lista fino a raggiungere la posizione dell'inserimento, o l'elemento da rimuovere, cosa che la classe *ArrayList* non è costretta a fare. Il tempo speso in tale operazione supera, nei casi esaminati dal test, il tempo necessario alla classe *ArrayList* per spostare gli elementi della lista. Notare che la classe *ArrayList* è leggermente più veloce della classe *LinkedList* anche nell'operazione di rimozione tramite iteratore, in cui entrambe le classi devono percorrere tutta la lista fino a raggiungere l'elemento da eliminare. Ciò è dovuto al fatto che l'operazione *next()* dell'iteratore associato alla classe *ArrayList* corrisponde al un semplice incremento di una variabile, che è una operazione più veloce rispetto a quella analoga dell'iteratore associato alla classe *LinkedList*.

A. File jar

Una applicazione Java è in genere costituita da un certo numero di classi. Tale numero, anche nel caso di applicazioni di media complessità, può divenire piuttosto elevato, rendendo compicata la distribuzione dell'applicazione agli utenti finali.

La piattaforma Java è dotata di uno strumento, il programma *jar*, che consente di raggruppare piuù classi in un unico archivio (file con estensione *.jar*). All'interno di un archivio jar possono essere poste, oltre alle classi, eventuali altre risorse necessarie all'applicazione, quali per esempio file immagine o file audio. Questo consente di implacchettare in un solo file una intera applicazione semplificandone notevolmente la fase di distribuzione. Nel caso degli applet, ciò riduce anche i tempi di scaricamento in quanto è sufficiente una singola richiesta HTTP (anzichè avere tante richieste quante sono le classi e le risorse che la compongono).

In genere un archivio jar contiene anche un file manifesto: un file all'interno del quale sono memorizzate meta-informazioni relative all'archivio o al suo contenuto nella forma di coppie *nome: valore*.

Il formato con cui il programma jar può essere mandato in esecuzione è il seguente:

jar [opzioni] [filejar] [manifesto] [inputfiles]

dove *filejar* è l'eventuale file archivio, *manifesto* il file manifesto, e *inputfiles* l'elenco dei file e/o delle cartelle da aggiungere/estrarre.

Le opzioni principali che possono essere specificate, quando il programma jar viene lanciato da riga di comando, sono le seguenti:

c Crea un nuovo archivio che contiene quanto specificato da *inputfiles*. Se è presente anche l'opzione *f* l'archivio viene memorizzato nel file specificato, altrimenti viene inviato sulla uscita standard.

f Specifica il nome del file che contiene l'archivio.

x Estrae tutti i file e le cartelle contenute nell'archivio (o estrae solo quanto specificato da *inputfiles* se tale parametro è presente). Se è presente anche l'opzione *f* l'estrazione avviene dal file specificato, altrimenti dall'ingresso standard.

t Mostra a video la lista dei file e delle cartelle contenute nell'archivio (o mostra solo quanto specificato da *inputfiles* se tale parametro è presente). Se è presente anche l'opzione *f* l'operazione avviene prendendo in considerazione il file specificato, altrimenti dall'ingresso standard.

u Aggiunge all'archivio i file o le cartelle specificate da *inputfiles*. Se è presente anche l'opzione *f* l'operazione avviene prendendo in considerazione il file specificato, altrimenti dall'ingresso standard.

m Include nell'archivio il file manifesto specificato.

Quando sono presenti piò opzioni gli argomenti di ognuna di esse devono apparire nello stesso ordine delle opzioni.

Alcuni esempi:

jar cf mioarchivio.jar *.class
Crea un archivio di nome *mioarchivio.jar* che contiene tutti i file *class* presenti nella cartella corrente. Include nell'archivio anche un file manifesto creato automaticamente.

jar cmf miomanifesto mioarchivio.jar prova/
Crea un archivio di nome *mioarchivio.jar* che contiene tutti i file presenti nella cartella *prova*. Include nell'archivio anche il file manifesto specificato.

jar xf mioarchivio.jar
Estrae tutti i file contenuti in *mioarchivio.jar*.

jar xf mioarchivio.jari file1 file2
Estrae *file1* e *file2* dall'archivio *mioarchivio.jar*.

Un archivio *jar* può essere parte del classpath della Java Virtual Machine. Per esempio:

java -classpath mioarchivio.jar MiaClasse
Manda in esecuzione il metodo *main()* di *MiaClasse* caricandola dall'archivio *mioarchivio.jar*. Da tale archivio verranno inoltre caricate le altre eventuali classi riferite da *MiaClasse*.

Un archivio *jar* può anche essere parte del classpath del compilatore:

javac -classpath lib1.jar Prova.java
Compila il file *Prova.java* cercando eventuali altre classi necessarie all'interno di *lib1.jar*.

Infine, l'interprete Java può essere invocato con l'opzione *-jar* che manda in esecuzione il programma contenuto nell'archivio specificato. Affinché ciò sia possibile è necessario che il manifesto contenuto nell'archivio sia dotato di una entrata
Main-Class: classname
che indica quale è la classe che deve essere usata come punto di ingresso per l'esecuzione dell'intero programma (tale classe deve essere dotata di un metodo *main()*).

Per esempio supponiamo che nella cartella corrente sia contenuta una sottocartella *miopack*, che a sua volta contiene un file *Prova.java*. Supponiamo inoltre che tale file contenga la definizione di un classe *Prova* che costituisce la classe principale della nostra applicazione. In questo caso è necessario preparare un file manifesto che contiene la riga
Main-Class: miopack.Prova
Il file manifesto, con nome *manifest.mf*, supponiamo che sia contenuto direttamente nella cartella corrente. È possibile creare l'archivio attraverso il seguente comando (emesso a partire dalla cartella corrente):
jar cmf manifest.mf mioarch.jar miopack/
A questo punto l'applicazione può essere mandata in esecuzione come segue:
java -jar mioarch.jar
Nel caso in cui l'applicazione necessiti di ulteriori classi e/o file, questi devono essere inclusi nell'archivio *mioarch.jar*.