Laboratori sul linguaggio Java

Mario G. Cimino, G. Dini

Pisa, 2005

Il presente materiale è stato prodotto durante l'attività didattica per il Corso di Tecnologie Informatiche per la Gestione Aziendale (CdL in Ing. Informatica per la Gestione d'Azienda) degli a.a. 2003-04, e 2004-05.

ESERCITAZIONE TIGA: Concetti e costrutti base di Java

1) Scrivere un programma *AnalizzaVoti* in JAVA che, data in ingresso una **sequenza di voti** $v_1\ v_2\ \ldots\ v_N$ (tra 18 e 33), la **visualizzi** sullo **schermo**, ne calcoli il **minimo**, il **massimo**, la **media** $\left(\bar{v} = \frac{\sum_{i=1}^{N} v_i}{N} \right)$ e la **variabilità** $\left(\tilde{v} = \frac{\sum_{i=1}^{N} |v_i - \bar{v}|}{N} \right)$.

```
D:\TIGA\lab1>java AnalizzaVoti
Occorre inserire almeno due voti separati da spazi

D:\TIGA\lab1>java AnalizzaVoti 18 33
voti:           18 33
voto minimo:    18
voto massimo:   33
media:          25.5    (buona)
variabilita':   7.5     (alta)

D:\TIGA\lab1>java AnalizzaVoti 18 33 20 24 25 30 22
voti:           18 33 20 24 25 30 22
voto minimo:    18
voto massimo:   33
media:          24.57   (buona)
variabilita':   4.08    (normale)

D:\TIGA\lab1>java AnalizzaVoti 33 33 33 33 33
voti:           33 33 33 33 33
voto minimo:    33
voto massimo:   33
media:          33.0    (eccellente)
variabilita':   0.0     (nessuna)
```

Fig.1 – Possibili scenari di esecuzione del programma AnalizzaVoti

2) Visualizzare anche la seguente **valutazione qualitativa** della media e della variabilità:
media: *sufficiente* in [18, 21), *discreta* in [21, 24), *buona* in [24, 27), *distinta* in [27, 30), *ottima* in [30, 33), *eccellente* in [33]
variabilità: *nessuna* in [0], *bassa* in (0, 2.5], *normale* in (2.5, 5.0], *alta* in (5.0, 7.5]).

3) Visualizzare i numeri con due sole cifre decimali.

Suggerimenti

- Si adoperi la medesima sintassi del C++ per le istruzioni di controllo (for, while, switch, do, ...), con i seguenti accorgimenti

- Il file **AnalizzaVoti.java** può essere cosi strutturato:

```
public class AnalizzaVoti {
    public static void main (String[] args) {

        // qui inserire tutte le istruzioni;  i voti in ingresso si trovano nell'array di stringhe args[]
        // e si possono leggere e convertire in interi con l'istruzione Integer.parseInt(stringa):
                int sommaVoti = 0;
                for ( int i = 0; i < args.length; i++ )
                        sommaVoti += Integer.parseInt(args[i]);
        // oppure stampare sul video con l'istruzione System.out.println(stringa)
                System.out.println (" primo  voto: " + args[0]);
                System.out.println (" ultimo voto: " + args[args.length-1]);
    }
}
```

- Per compilare digitare "**`javac AnalizzaVoti.java`**" dal prompt dei comandi.
- Per eseguire digitare "**`java AnalizzaVoti`**" seguito dalla sequenza di voti separati da spazi (come in Fig.1).
- Altri metodi utili della classe Math (simili alle funzioni della libreria <math> del C++): **`Math.min(x,y)`, `Math.max(x,y)`, `Math.floor(x)`, `Math.ceil(x)`, `Math.round(x)`, `Math.abs(x)`, `Math.pow(x,y)`.**

ESERCITAZIONE TIGA: Concetti e costrutti base di Java - SOLUZIONE PROPOSTA

```
// AnalizzaVoti.java
public class AnalizzaVoti {
    public static void main (String[] args) {
        final int       PRECISIONE = 2;
        final double    FATTORE = Math.pow(10,PRECISIONE);
        final int       MIN = 18;
        final int       MAX = 33;
        final float     VARIABILITA_MAX = 7.5F;

        // controllo iniziale
        if (args.length < 2) {
            System.err.println("Occorre inserire almeno due voti separati da spazi");
            System.exit(1);
        }

        // stampa della sequenza inserita
        System.out.print ("voti:\t\t");
        for ( int i = 0; i < args.length; i++)
            System.out.print (args[i] + ' ');
        System.out.println ();

        // calcolo del massimo e minimo
        int  min = Integer.parseInt (args[0]),
             max = Integer.parseInt (args[0]);
        int i = args.length-1;
        for ( ; ; ) {
            min = Math.min(min, Integer.parseInt (args[i]) );
            max = Math.max(max, Integer.parseInt (args[i]) );
            i--;
            if (i==0)
                break;
        }
        System.out.println ("voto minimo:\t" + min + "\nvoto massimo:\t" + max );

        // calcolo del valor medio
        i = 0;
        float media = 0;
        while (i < args.length) {
            media += Integer.parseInt (args[i]);
            i++;
        }
        media /= args.length;
        System.out.print ("media:\t\t" + Math.round(media*FATTORE)/FATTORE );

        // valutazione qualitativa
        switch((int)Math.floor((media-MIN)/(MAX-MIN)*5)) {
            case 0:   System.out.println ( "\t(sufficiente)" );   break;
            case 1:   System.out.println ( "\t(discreta)" );      break;
            case 2:   System.out.println ( "\t(buona)" );         break;
            case 3:   System.out.println ( "\t(distinta)" );      break;
            case 4:   System.out.println ( "\t(ottima)" );        break;
            default:  System.out.println ( "\t(eccellente)" );
        }
        // calcolo della variabilita'
        float variabilita = 0;
        int x;
        i = 0;
        do {
            x = Integer.parseInt (args[i]);
            variabilita += Math.abs (x-media);
            i++;
        }
        while (i < args.length);
        variabilita /= args.length;
        System.out.print ("variabilita':\t" + Math.round(variabilita*FATTORE)/FATTORE );

        // valutazione qualitativa
        switch((int)Math.ceil(variabilita/VARIABILITA_MAX*3)) {
            default:  System.out.println ( "\t(nessuna) " );  break;
            case 1:   System.out.println ( "\t(bassa)" );     break;
            case 2:   System.out.println ( "\t(normale)" );   break;
            case 3:   System.out.println ( "\t(alta)" );
        }
    } // main
  } // class
```

ESERCITAZIONE TIGA: Classi ed oggetti

Una *banca* gestisce conti bancari per conto dei propri clienti. Ogni cliente è titolare di un solo conto. Ciascun conto ha un solo titolare. Un conto bancario è caratterizzato da un numero di conto e da un saldo. Il numero di conto identifica univocamente un conto all'interno della banca. In accordo alle politiche della banca, il saldo di un conto bancario non può essere negativo. Un cliente è caratterizzato da un nome, un numero di cliente, un PIN e dal conto bancario di cui è titolare. Il numero di cliente identifica univocamente il cliente.

Le operazioni che possono essere fatte su di un ***conto bancario*** sono (almeno) le seguenti:

- `public Conto()` costruttore di default che assegna a questo conto un numero ed imposta il saldo a zero.
- `public Conto(double saldoIniziale)` costruttore che assegna a questo conto un numero ed imposta il saldo ad un valore iniziale specificato da `saldoIniziale`.
- `public void deposito(double quanto)` che incrementa il saldo di questo conto bancario della quantità specificata da `quanto`.
- `public boolean prelievo(double quanto)` che, se possibile, preleva da questo conto bancario una quantità specificata da `quanto`. Il metodo ritorna il valore `true` se il prelievo è stato effettuato; il valore `false` altrimenti.
- `public double ritornaSaldo()` che ritorna il valore corrente del saldo di questo conto bancario.
- `public boolean trasferisci(Conto altro, double quanto)` che trasferisce, se possibile, una quantità pari a `quanto` da questo conto al conto specificato da `altro`. Il metodo ritorna il valore `true` se il trasferimento è stato effettuato; il valore `false` altrimenti.

Le operazioni che possono essere fatte su di un ***cliente*** sono (almeno) le seguenti:

- `public Cliente (String nome, int numCliente, int pin, Conto conto)` che crea un cliente con nome, numero cliente, pin e conto pari a `nome`, `numCliente`, `pin` e `conto` rispettivamente.
- `public boolean verifica (int numCliente, int pin)` che verifica se `numCliente` e `pin` sono, rispettivamente, il numero di cliente ed il PIN di questo cliente.
- `public int ritornaNumCliente()` che ritorna il numero cliente di questo cliente.

Le operazioni che possono essere fatte sulla ***banca*** sono (almeno) le seguenti:

- `public Banca()` che imposta il numero massimo di clienti di questa banca ad un valore iniziale predefinito pari a `DEFAULT_MAX_TOT_CLIENTI`.
- `public Banca(int maxTotClienti)` che imposta il numero massimo di clienti di questa banca ad un valore iniziale specificato da `maxTotClienti`. Se tale parametro specifica un valore minore di `1`, il costruttore imposta il numero massimo di clienti al valore iniziale predefinito.
- `public boolean registra(Cliente cliente)` che include, se possibile, a questa banca un nuovo cliente specificato da `cliente`, ritornando il valore `true`, il valore `false` altrimenti.
- `void cancella(Cliente cliente)` che cancella dalla banca il riferimento al cliente specificato da `cliente`.
- `public Cliente trova(int numCliente)` che restituisce il cliente di questa banca il cui numero cliente è uguale a quello specificato da `numCliente`. Il metodo torna `null` se tale cliente non è presente in questa banca.
- `public Cliente trova(String nome)` che ritorna il cliente di questa banca il cui nome è uguale a quello specificato da `nome`. Il metodo torna `null` se tale cliente non è presente in questa banca.
- `public boolean trasferisci(int numClienteSrg, int numClienteDst, double quanto)` che trasferisce la quantità `quanto` tra i conti bancari di due clienti di questa banca. I due clienti sono specificati per mezzo del loro numero di cliente. La quantità viene trasferita dal conto del cliente `numClienteSrg` al conto del cliente `numClienteDst`.

ESERCITAZIONE TIGA: Classi ed oggetti – SOLUZIONE PROPOSTA

Progettazione del sistema *Banca*:

• In Fig.1 ogni "scatola" rappresenta una classe, divisa in tre sezioni: il nome, gli attributi (campi), ed i metodi (funzioni), questi ultimi non indicati per semplicità. I fogli dall' angolo ripiegato contengono note importanti o vincoli relativi agli elementi riferiti con la linea tratteggiata.

Nella schematizzazione proposta, il *Cliente* è caratterizzato dagli attributi *nome*, *numero*, *pin* e *Conto*, quest' ultimo raggiungibile attraverso un riferimento (la freccia); mentre *Conto* non riferisce il relativo *Cliente* (sarebbe superfluo, perché nelle specifiche non è richiesto di individuare il *Cliente*, dato il suo *Conto*) quindi manca la freccia di ritorno; le due classi hanno una relazione "1 ad 1".
Il rombo pieno indica *composizione*: l'oggetto composto *Cliente* gestisce in modo esclusivo il proprio oggetto componente *Conto*. Il rombo vuoto indica qualcosa di meno restrittivo, l'*aggregazione*: un oggetto aggregato *Cliente* può avere una esistenza indipendente dall'oggetto aggregatore *Banca* che non può gestirlo in modo esclusivo come un proprio componente. Ad esempio, in un modello più esteso alcuni clienti potrebbero essere aggregati di un' altra classe che offre servizi aggiuntivi. Inoltre, *Banca* può contenere da zero a molti *Clienti*, e questi sono riferiti da un solo oggetto *Banca* (un cliente di due banche deve essere rappresentato in due diversi oggetti *Cliente*).

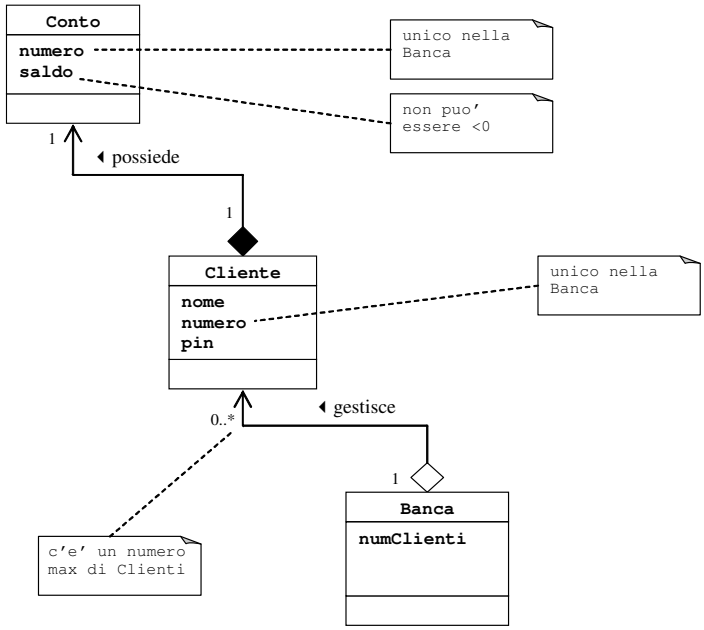


Fig.1 – Diagramma delle classi (semplificato) del sistema *Banca*.

Scelte implementative:

• Nel principale intento di introdurre all'uso del costrutto *class*, la soluzione proposta per *Banca* utilizza gli Array senza ordinamento, quindi con ricerca esaustiva, e non adopera le utilità presenti nella classe `Arrays` (ordinamento, ricerca, ...) delle API[1] di Java. Le API contengono numerose classi che verrano introdotte man mano nelle prossime esercitazioni, come `Vector` (array di dimensione variabile), *`Collection`*, *`List`*, *`Set`*, *`Map`*, etc. che implementano tutte le tipiche strutture dati dell'informatica, ed altre ancora, come *`JFrame`*, *`JButton`*, … che implementano i componenti di una interfaccia grafica (pannelli, bottoni, …). Sono classi facilmente utlizzabili ed hanno metodi dai nomi molto chiari e semplici; la filosofia di Java aderisce ottimamente ai principi dell'Ingegneria del Software: progettare un sistema software adoperando componenti preesitenti, e limitando l'implementazione di funzionalità "ad hoc"; in tal modo si scrive meno codice, più riusabile e modificabile perchè le API costituiscono uno *standard* per la comunità dei programmatori Java.

• Le classi implementate non sono "blindate" da possibili valori inconsistenti in ingresso; ad esempio e' possibile creare un *Conto* con un `saldoIniziale` negativo, depositare una quantità negativa, prelevare una quantità negativa; questi controlli sono importanti nel mondo reale, perchè la loro assenza rende il software attaccabile dagli *hacker*; si possono realizzare efficientemente con tecniche apposite che verranno studiate, basate sulle *eccezioni*; infatti la soluzione di gestire le anomalie inserendo nel codice numerosi blocchi "if" è poco professionale, in quanto danneggia la leggibilità della funzione principale che il metodo deve implementare.

Codifica in linguaggio Java:

```
// Conto.java
public class Conto {

    public  Conto () {
        this(0);
    }

    public  Conto (double saldoIniziale) {
            saldo = saldoIniziale;
    }

    public void deposito (double quanto) {
            saldo += quanto;
    }
```

[1] Application Programming Interface (Interfaccia di Programmazione dell'Applicazione) è l'insieme di classi Java che il programmatore ha già a disposizione per sviluppare le applicazioni, documentate mediante pagine html di agile consultazione (http://java.sun.com/j2se/1.4.2/docs/api/index.html) .

```
    public boolean prelievo (double quanto) {
        if (saldo >= quanto) {
            saldo -= quanto;
            return true;
        }
        return false;
    }

    public double ritornaSaldo () {
        return saldo;
    }

    public int ritornaNumero () {
        return numero;
    }

    public boolean trasferisci (Conto altro, double quanto) {
        if (prelievo (quanto)) {
            altro.deposito (quanto);
            return true;
        }
        return false;
    }

    private static int  estraiNuovoNumero () {
        return ++nuovoNumero;
    }

    private double      saldo;
    private final int   numero = estraiNuovoNumero ();
    private static int  nuovoNumero = 0;
}

// Cliente.java

public class Cliente {

    public Cliente (String nome, int numCliente, int pin, Conto conto) {
        this.nome = nome;
        numero = numCliente;
        this.pin = pin;
        this.conto = conto;
    }

    public boolean verifica (int numCliente, int pin) {
        return (numero == numCliente) && (this.pin == pin);
    }

    public int ritornaNumCliente () {
        return numero;
    }

    public String ritornaNome () {
        return nome;
    }

    public double ritornaSaldo () {
        return conto.ritornaSaldo();
    }

    public boolean trasferisci(Cliente altro, double quanto) {
        return conto.trasferisci(altro.conto, quanto);
    }

    private String  nome;
    private int     numero;
    private int     pin;
    private Conto   conto;
}


// Banca.java

public class Banca {
    public  Banca () {
        elencoClienti = new Cliente[DEFAULT_MAX_TOT_CLIENTI];
    }

    public  Banca (int maxtotClienti) {
        if (maxtotClienti < 1)
           maxtotClienti = DEFAULT_MAX_TOT_CLIENTI;
        elencoClienti = new Cliente [maxtotClienti];
    }
```

```
public boolean registra (Cliente cliente) {
    if (totClienti == elencoClienti.length)
        return false;
    int i;
    for( i = 0; elencoClienti [i] != null; i++) ;
    elencoClienti[i] = cliente;
    totClienti++;
    return true;
}

public void cancella (Cliente cliente) {
    if (totClienti == 0)
        return;
    for (int i = 0; i < elencoClienti.length; i++)
        if (elencoClienti[i] == cliente) {
            totClienti--;
            elencoClienti[i] = null;
            break;
        }
}

public Cliente trova (int numCliente) {
    if (totClienti == 0)
        return null;
    for (int i = 0; i < elencoClienti.length; i++) {
        if (elencoClienti[i] == null)
            continue;
        if (elencoClienti[i].ritornaNumCliente() == numCliente)
            return elencoClienti[i];
    }
    return null;
}

public Cliente trova (String nome) {
    if (totClienti == 0)
        return null;
    for (int i = 0; i < elencoClienti.length; i++) {
        if (elencoClienti[i] == null)
            continue;
        if (elencoClienti[i].ritornaNome().equals(nome))
            return elencoClienti[i];
    }
    return null;
}

public boolean trasferisci (int numClienteSrg, int numClienteDst, double quanto) {
    Cliente clienteSrg = trova(numClienteSrg);
    if (clienteSrg == null)
        return false;
    Cliente clienteDst = trova(numClienteDst);
    if (clienteDst == null)
        return false;
    return clienteSrg.trasferisci(clienteDst, quanto);
}

private Cliente[] elencoClienti;
private int totClienti = 0;
private static final int DEFAULT_MAX_TOT_CLIENTI = 9;

public static void main(String[] args) {
    Banca bancoDiRoma = new Banca(2);

    if ((bancoDiRoma.registra (new Cliente ("Romolo", 101, 1234, new Conto() ))) &&
        (bancoDiRoma.registra (new Cliente ("Remo", 102, 2341, new Conto(1000.00) ))))

        System.out.println("Aggiunti i clienti " +
                           bancoDiRoma.trova(101).ritornaNome() + " e " +
                           bancoDiRoma.trova(102).ritornaNome() );

    if (!bancoDiRoma.registra (new Cliente ("Lupa", 100, 3412, new Conto() )))
        System.out.println("La banca ha raggiunto il limite massimo di clienti");

    if (bancoDiRoma.trova ("Lupa") == null)
        System.out.println ("Il cliente Lupa non e` registrato");

    Cliente donatore = bancoDiRoma.trova ("Remo"),
            ricevente = bancoDiRoma.trova ("Romolo");

    System.out.println (donatore.ritornaNome() + " ha il numero " +
                        donatore.ritornaNumCliente());

    System.out.println (donatore.ritornaNome() + " ha " +
```

```
                        donatore.ritornaSaldo() + " euro, " +
                        ricevente.ritornaNome() + " ha " +
                        ricevente.ritornaSaldo() + " euro.");

        double importo = 1000.00;
        if (donatore.trasferisci (ricevente, importo))
            System.out.println ("Trasferiti " + importo + " euro da " +
                                donatore.ritornaNome() + " a " +
                                ricevente.ritornaNome());

        System.out.println (donatore.ritornaNome() + " ha " +
                        donatore.ritornaSaldo () + " euro, " +
                        ricevente.ritornaNome() + " ha " +
                        ricevente.ritornaSaldo () + " euro.");

        if (!donatore.trasferisci (ricevente, importo))
            System.out.println ("Impossibile trasferire " + importo + " euro da " +
                                donatore.ritornaNome () + " a " + ricevente.ritornaNome ());

        int numeroDonatore = donatore.ritornaNumCliente();
        int numeroRicevente = ricevente.ritornaNumCliente();

        bancoDiRoma.cancella (donatore);
        if (bancoDiRoma.trova (numeroDonatore) == null)
            System.out.println ("Il cliente n. " + numeroDonatore + " non esiste");

        if (bancoDiRoma.trova (numeroRicevente) != null)
            System.out.println ("Il cliente n. " + numeroRicevente +
            " e` " + ricevente.ritornaNome ());

        } // main
} // class
```

Testing del programma:

• Per semplicità si è cercato di implementare solo i metodi strettamente necessari, per cui mancano ad esempio dei metodi per visualizzare lo stato di un oggetto, per leggere o scrivere su file un elenco di oggetti, per stampare messaggi di servizio o di errore agli utenti, etc..; tutti questi argomenti saranno trattati in successive esercitazioni. A causa di questa mancanza, nel metodo **main** occorre invocare direttamente i diversi metodi per stampare le informazioni richieste, per cui globalmente esso risulta poco leggibile e ridondante, da non prendersi a modello come stile di programmazione.

```
// output
Aggiunti i clienti Romolo e Remo
La banca ha raggiunto il limite massimo di clienti
Il cliente Lupa non e` registrato
Remo ha il numero 102
Remo ha 1000.0 euro, Romolo ha 0.0 euro.
Trasferiti 1000.0 euro da Remo a Romolo
Remo ha 0.0 euro, Romolo ha 1000.0 euro.
Impossibile trasferire 1000.0 euro da Remo a Romolo
Il cliente n. 102 non esiste
Il cliente n. 101 e` Romolo
```

ESERCITAZIONE TIGA: Ereditarietà, toString, equals, hashCode; javap, javadoc

In aggiunta a quanto specificato nella esercitazione su *Classi ed oggetti*:

a) La banca gestisce quattro diversi tipi di conto bancario
- Il *conto bancario* (`Conto`)
- Il *libretto di risparmio* (`LibrettoDiRisparmio`), un *conto bancario* che frutta interessi mensili.
- Il *conto corrente* (`ContoCorrente`), un *conto bancario* che non ha interessi, offre un numero limitato di operazioni mensili gratuite ma addebita una commissione per ciascun movimento aggiuntivo.
- Il *conto vincolato* (`ContoVincolato`) che è un *libretto di risparmio* che impegna a lasciare il denaro nel conto per certo numero di mesi ed prevede una penale per il ritiro anticipato.

b) Sul *conto bancario* possono compiersi anche le seguenti operazioni:
- `public boolean equals(Object obj)` che sovrascrive l'omonimo metodo predefinito per la classe `Object`; restituendo `true` se i due conti hanno saldo identico, `false` altrimenti;
- `public int hashCode()` che sovrascrive l'omonimo metodo predefinito per la classe `Object`, restituendo un intero sulla base del valore del campo `saldo`.

c) Definire analogamente i metodi `equals` e `hashCode` sul *cliente*, basandosi esclusivamente sui risultati dei metodi definiti al punto b)

d) Sulla *banca* possono compiersi anche le seguenti operazioni:
- `public void aggiornamenti()` che esegue le operazioni periodiche previste per ciascun tipo conto:
  - aggiunge gli interessi mensili ai *libretti di risparmio*;
  - aggiunge gli interessi ai *conti vincolati*;
  - preleva le commissioni dai *conti correnti*.
- `public String toString()` che restituisce una rappresentazione testuale (intelligibile) dello stato interno dell'oggetto come stringa, sovrascrivendo il metodo predefinito per la classe `Object`, con il seguente formato:

```
nomeClasse [
 nomeCampo1 = valoreCampo1;
 ...
 nomeCampoN = valoreCampoN;
]
```

  dove `nomeCampo` può essere un attributo *proprio* della classe, ovvero una *superclasse* o una *classe contenuta*.

e) Si adoperino i seguenti strumenti a riga di comando della directory *bin* (digitare `-help` dopo il comando per i dettagli)*:*

- *javac* per compilare assieme diversi files di cui sia noto il nome.
- *javadoc* per generare automaticamente la documentazione html
- *javap* per visualizzare i metodi, i campi, o il bytecode di un file `.class`

ESERCITAZIONE TIGA: Ereditarietà, toString, …, javadoc – SOLUZIONE PROPOSTA

Progettazione:

• In Fig.1 abbiamo aggiunto (al diagramma dell'esercitazione su *Classi ed Oggetti*) le tre classi *LibrettoDiRisparmio, ContoCorrente e ContoVincolato*, che rappresentano *estensioni*, ossia includono caratteristiche aggiuntive alla superclasse *Conto*. Nel verso indicato dal triangolo vuoto, da sottoclasse a superclasse, si ha quindi una *riduzione* [2].

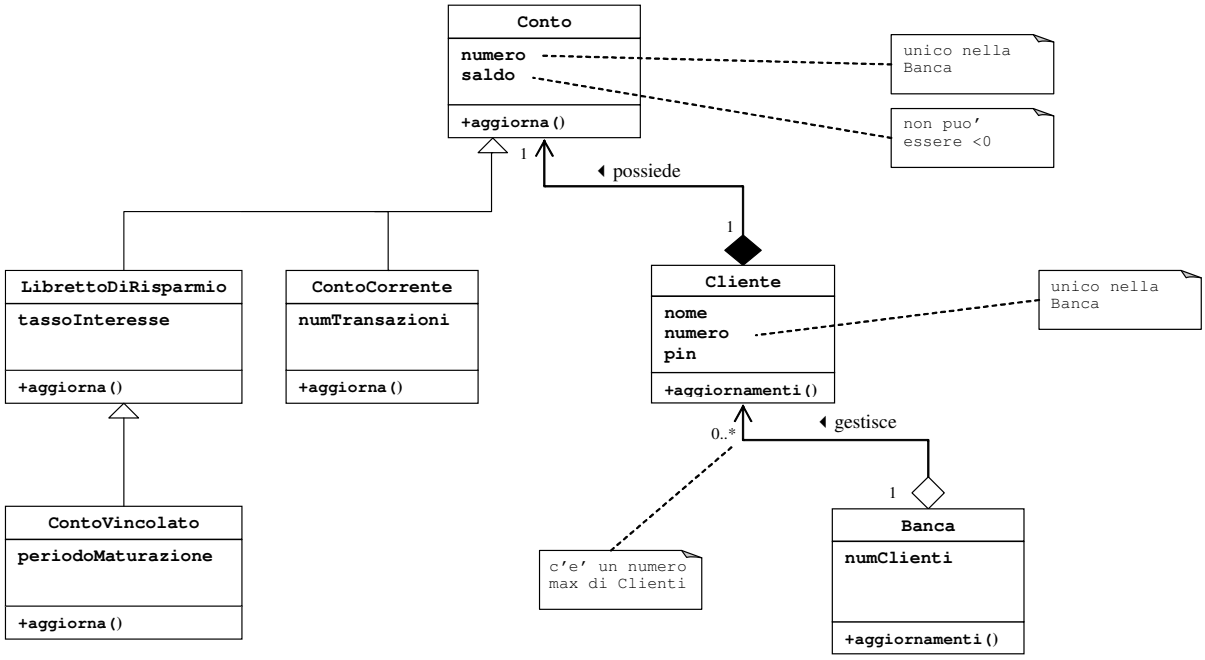


Fig.1 – Diagramma delle classi (semplificato) del sistema *Banca.*

Il metodo *aggiorna()* è stato inserito sia in *Conto* che in tutte le sue sottoclassi; il simbolo "+" indica che il metodo è *public* [3].

Il problema che ci poniamo è: chi decide il metodo *aggiorna*() da invocare sul *Cliente* a seconda del tipo di *Conto* che possiede ?

Ad esempio, consideriamo un cliente con un conto corrente:

```
Cliente questoCliente = new Cliente ("Romolo", 102, 2341, new ContoCorrente (100)));
```

invocando il metodo *aggiorna*() su questo cliente, dovrebbe avvenire qualcosa del genere:

```
if <questoCliente ha un ContoCorrente>
        questoCliente.aggiornamentiContoCorrente();
else if <questoCliente ha un ContoVincolato>
        questoCliente.aggiornamentiContoVincolato()
...
```

dove le diverse versioni del metodo *aggiornamenti()* eseguite su un cliente sono state rinominate per chiarezza. Chiaramente questo non è professionale, vediamo perchè.

Innanzitutto, nè la classe *Cliente* nè *Conto* devono gestire questa differenziazione, poichè in pratica possono ramificarsi decine di estensioni, e ciò farebbe aumentare troppo la complessità della classe e diminuirne la manutenibilità; inoltre è possibile che non si abbia il codice sorgente di *Conto* (il file *Conto.java*) ma solo il bytecode (il file *Conto.class*) per cui non si possa aggiungere del codice a *Conto*; idem per *Cliente*.

La soluzione consiste nel mantenere in *Cliente* il riferimento alla superclasse *Conto*, sulla quale definiamo un metodo *aggiorna()* dal corpo vuoto [4]; dopodichè ridefiniamo *aggiorna*() in ogni sottoclasse. Nel metodo

```
questoCliente.aggiornamenti()
```

vi sarà solo un'istruzione `conto.aggiorna()` che eseguirà automaticamente il metodo *aggiorna()* specifico del tipo di conto riferito, anche se il riferimento nella classe *Cliente* è genericamente a *Conto* [5].

Quindi, il fatto di aver individuato una relazione di ereditarietà in *Conto* ha permesso di semplificare la manutenzione del codice e di adoperare il medesimo metodo in tutta la gerarchia: l'importanza di adoperare le classi e le relazioni tra di esse (ereditarietà, composizione, aggregazione, ...) è dovuta quindi al supporto dato dal compilatore, oltre che alla chiarezza con cui si modella la realtà (qualsiasi programma è un modello di qualche aspetto della realtà).

---

[2] Per fissare le idee, si osservi che l'ampiezza del triangolo si *riduce* dalla base al vertice.

[3] Si indica *private* con "-", e *protected* con "#", e ciò si può specificare sia per i metodi che per i campi).

[4] Potrebbe esserci anche del codice, che però verrà ignorato se non invocato esplicitamente dalla sottoclasse mediante *super.aggiorna()*.

[5] Tutti i conti specifici sono comunque dei conti.

In generale, la programmazione ad oggetti consente: lo sviluppo di programmi modulari (suddivisione del lavoro tra i dipendenti), la modificabilità (cambio internamente la classe, senza modificare i metodi *pubblici*, ossia i servizi offerti alle altre classi), la semplificazione dello sviluppo (come visto nell'esempio di *aggiorna()*), la cooperazione tra gruppi e la integrità concettuale del sistema (tutti adoperano il medesimo costrutto *classe*, sufficientemente generale da adattarsi ai vari concetti mentali identificabili nel mondo reale). Occorre tener presente che i software industriali sono composti da decine o centinaia di classi, mentre ciascun programmatore ha sempre a disposizione una sola mente. Le gerarchie, le relazioni, i metodi, concetti apparentemente ingombranti per piccoli software, servono a ridurre la complessità man mano che il software cresce (vedremo ad esempio anche la gerarchia di visibilità costruita con i *package*, la cui utilità è evidente quando si sfoglia la pagina delle API).

Suggerimenti pratici per la progettazione ad oggetti:

1) Come si "vedono" le classi dal mondo reale ? Una buona classe non deve solo rappresentare un concetto mentale ben definito, ma anche un insieme di metodi *minimo* (deve esporre come *public* la minima quantità necessaria di informazione). La domanda da porsi è: "Di quale entità è rilevante mantenere l'informazione ?. Spesso risulta agevole affidarsi al vocabolario adoperato dagli esperti della materia (i quali hanno in mente dei concetti ben sedimentati), quindi il problema *induttivo* di "vedere" le classi, diventa *linguistico-deduttivo*: ad esempio, è buona norma nominare i campi con i *sostantivi*, i metodi con i *verbi* (perchè sono delle azioni), le classi con nomi *comuni*, gli oggetti con nomi *propri*, tutto eventualmente seguito da *aggettivi*.

2) Come si sceglie tra *contenimento* ed *ereditarietà* ? Derivare *Aereo* da *Motore* non va bene, perchè un *Aereo* non è un *Motore*, semmai ha un *Motore*. Un modo per vederlo è pensare se *Aereo* può avere due o più *Motore*. Quindi *Cliente* ha un *Conto* (averne due nella fattispecie è vietato ma non è assurdo in generale), mentre *ContoCorrente* è un (tipo particolare di) *Conto*.

3) Come si sceglie tra i due tipi di contenimento, la *composizione* e l'*aggregazione* ? La *composizione* è una particolare associazione di *aggregazione* con il vincolo che, ad ogni istante, un oggetto della classe componente deve essere parte di un solo oggetto della classe composta, quindi il comportamento di quest'ultima può essere progettato senza la conoscenza di altre classi che potrebbero creare o distruggere il componente. Un oggetto *Cliente* può gestire in modo esclusivo l'oggetto *Conto* come se fosse un proprio attributo, ma più complesso. Altri esempi sono *Firma* componente di *Contratto*, o *Casella* componente di *Dama*.

4) Come si individuano gli *attributi* di una classe ? Rispondendo alla domanda: "Che cosa è *necessario* conoscere di un oggetto della classe X ai fini della realtà che il nostro software deve gestire?.

Note sugli strumenti "a riga di comando" della directory *bin*

Il vero scopo delle istruzioni a riga di comando non è quello di essere adoperate "a mano", ma di permettere la creazione di file *script* che automatizzino procedure ripetitive o altrimenti complesse (ad esempio una ricompilazione *intelligente* di centinaia di classi, che prenda in considerazione solo i file *.java* che hanno una data posteriore ai corrispondenti *.class*). Un ambiente che offre la possibilità di invocare da Shell (interprete dei comandi) gli strumenti di sviluppo primitivi è quindi più professionale e versatile. In questa sede, l'utilizzo di tali strumenti ha lo scopo di introdurre in modo diretto al funzionamento dei componenti della Java Platform, pertanto vengono forniti ulteriori cenni prima di passare all' ambiente integrato di sviluppo (IDE) netBeans.

Se si ha a disposizione solo il file Conto.class (il bytecode), perché chi ha sviluppato la classe non vuole adottare la filosofia *open source*, ad es. per custodire un trucco geniale, o evitare che un *hacker* individui nel codice dei punti di attacco, si digita:

```
javap Conto
```

ed appare l'elenco dei metodi pubblici della classe. Per vedere il bytecode digitare `javap -c Conto`.

In ambienti Windows © può essere necessario inizializzare la variabile di ambiente CLASSPATH con il comando:

```
set CLASSPATH=
```

per far funzionare `java o javac`.

Per compilare molti file assieme ci sono diverse alternative, mostrate con i seguenti esempi:

1) `javac Cliente.java Conto.java Banca.java`
2) `javac *.java`
3) `javac @mialista`

dove mialista è un file di testo contenente l'elenco delle classi da compilare

```
Conto.java
Cliente.java
Banca.java
```

Per ulteriori dettagli digitare il comando seguito da "-help", es.

```
java  -help
javac -help
javap -help
```

Esiste anche il debugger (comando `jdb`) ed il generatore di documentazione (comando `javadoc`). Infine, un visualizzatore di *applet* (comando `appletviewer`) che verrà usato in successive esercitazioni. Un'applet è una piccola applicazione Java che non parte da sè ma va integrata in un' altra applicazione, ad esempio, come componente in una pagina web. L'applet viene eseguita dalla Java Virtual Machine del browser, precedentemente installata come plug-in[6]; un altro tipo di applicazione Java è la *servlet* , che viene eseguita dalla macchina virtuale di un *application server*, ad esempio per generare pagine web in modo dinamico.

[6] I plug-in sono dei programmi che estendono l'usabilità di un'applicativo. Per i web browser vi sono plug-in per eseguire file audio *Real Time* ®, file video, animazioni *Flash Macromedia* ®, *Java Applet* ®, e tanti altri.

Codifica in linguaggio Java:

Per i file della esercitazione su *Classi ed oggetti*, sono indicate solo le righe di codice da aggiungere o sostituire.

```
// Conto.java
(aggiungere, es. prima delle definizioni private)
public void aggiorna() { }

    public String toString () {
        return "\n" + getClass().getName() + " [\n saldo = " + saldo + "\n] ";
    }

    public boolean equals(Object obj) {
        if ( !(obj instanceof Conto) )
            return false;
        return ((Conto)obj).saldo == saldo;
    }

    public int hashCode() {
        return (int)saldo;
}

// ContoCorrente.java
public class ContoCorrente extends Conto {

    public ContoCorrente (double saldoIniziale) {
        super (saldoIniziale);
        totTransazioni = 0;
    }

    public void deposito (double quanto) {
        totTransazioni++;
        super.deposito (quanto);
    }

    public boolean prelievo (double quanto) {
        totTransazioni++;
        return super.prelievo (quanto);
    }

    public void aggiorna () {
        if (totTransazioni > TRANSAZIONI_GRATUITE) {
            double commissioni = COMMISSIONE_TRANSAZIONE *
                                 (totTransazioni - TRANSAZIONI_GRATUITE);
            super.prelievo(commissioni);
        }
        totTransazioni = 0;
    }

    public String toString() {
        return  "\n" + getClass().getName() + " [\n totTransazioni = " +
                totTransazioni + ";\n" +
                super.toString() +
                "\n] ";
    }

    private int totTransazioni;
    private static final int TRANSAZIONI_GRATUITE = 3;
    private static final double COMMISSIONE_TRANSAZIONE = 2.0;

}

// LibrettoDiRisparmio.java
public class LibrettoDiRisparmio extends Conto {

    public LibrettoDiRisparmio (double tasso) {
        tassoInteresse = tasso;
    }

    public void aggiorna () {
        double interessi = ritornaSaldo () * tassoInteresse / 100;
        deposito (interessi);
    }

    public String toString() {
        return  "\n" + getClass().getName() + " [\n tassoInteresse = " +
                tassoInteresse + ";\n" +
                super.toString() +
                "\n] ";
    }

    private double tassoInteresse;
}

// ContoVincolato.java
public class ContoVincolato extends LibrettoDiRisparmio {

    public ContoVincolato (double tasso, int maturazione) {
        super (tasso);
        periodoMaturazione = maturazione;
    }

    public void aggiorna () {
        periodoMaturazione--;
        super.aggiorna();
```

```
    }

    public boolean prelievo (double quanto) {
        boolean result = true;
        if (periodoMaturazione > 0)
            result = super.prelievo (PENALE_PER_PRELIEVO_ANTICIPATO);
        if (result)
            result = super.prelievo (quanto);
        return result;
    }

    public String toString() {
        return  "\n" + getClass().getName() + " [\n periodoMaturazione = " +
                periodoMaturazione + ";\n" +
                super.toString() +
                "\n] ";
    }

    private int periodoMaturazione;
    private static double PENALE_PER_PRELIEVO_ANTICIPATO = 20;
}


// Cliente.java
(aggiungere, es. prima delle definizioni private)
    public void aggiornamenti() {
        conto.aggiorna();
    }

    public String toString() {
        return  "\n" + getClass().getName() + " [\n nome = " + nome +
                ";\n numero = " + numero +
                ";\n pin = " + pin + "; " +
                conto.toString() +
                "\n] ";
    }

    public boolean equals(Object obj) {
        if ( !(obj instanceof Cliente) )
            return false;
        return conto.equals(((Cliente)obj).conto);
    }

    public int hashCode() {
        return conto.hashCode();
    }

// Banca.java
(aggiungere, es. prima delle definizioni private)
    public void aggiornamenti() {
        for (int i=0; i < elencoClienti.length; i++)
            if (elencoClienti[i] != null)
                elencoClienti[i].aggiornamenti();
    }

    public String toString() {
        String risultato = "\n" + getClass().getName() +
                           " [\n totClienti = " + totClienti + ";";
        for (int i=0; i < elencoClienti.length; i++)
            if (elencoClienti[i] != null)
                risultato += elencoClienti[i].toString();
        return risultato + "\n] ";
    }

(per il testing, riscrivere il metodo main es. come segue)
    public static void main(String[] args) {
        Banca bancoDiRoma = new Banca();

        bancoDiRoma.registra (new Cliente ("Romolo", 101, 1234, new Conto(100.00) ));
        bancoDiRoma.registra (new Cliente ("Remo"  , 102, 2341, new ContoCorrente(100.00) ));
        bancoDiRoma.registra (new Cliente ("Lupa"  , 103, 3412, new LibrettoDiRisparmio(1.5)));
        bancoDiRoma.registra (new Cliente ("Lupo", 104, 4123, new ContoVincolato(3.0, 12)));

        Cliente donatore = bancoDiRoma.trova("Romolo");
        Cliente ricevente = bancoDiRoma.trova("Lupa");

        for (int i=0; i < 3; i++)
            donatore.trasferisci(ricevente, 10.00);

        donatore = bancoDiRoma.trova("Remo");
        ricevente = bancoDiRoma.trova("Lupo");

        for (int i=0; i < 5; i++)
            donatore.trasferisci(ricevente, 10.00);
        ricevente.trasferisci(donatore, 10.00);

        bancoDiRoma.aggiornamenti();
        System.out.println(bancoDiRoma.toString());

        donatore.trasferisci(ricevente, 16.70);
        System.out.println(donatore.equals(ricevente));

    } // main
```

Testing del programma[7]:

- Innanzitutto si osservi che, anche se il metodo *trasferisci* è definito esclusivamente nella classe base, i metodi *prelievo* e *deposito* da esso invocati sono sempre corrispondenti al tipo specifico di conto. In base a ciò:
  a) l'oggetto *Conto* del cliente "Romolo", di saldo iniziale *€ 100.00*, è soggetto a 3 prelievi di *€ 10.00*, per cui il saldo finale risulta *€ 70.00*.
  b) in corrispondenza ad a), l'oggetto *LibrettoDiRisparmio* del cliente "Lupa", di saldo iniziale zero, è soggetto a 3 versamenti di importo *€ 10.00*, per un totale di *€ 30.00*, a cui vanno aggiunti gli interessi al tasso *1.5%*, pari a *€ 30.00 * 0.015 = € 0.45*.
  c) l'oggetto *ContoCorrente* del cliente "Remo", di saldo iniziale *€ 100.00*, è soggetto a 5 prelievi di *€ 10.00*, ed un versamento di *€ 10.00,* per un prelievo complessivo di *€ 40.00* in 6 transazioni, di cui 3 a pagamento (*3 * € 2.00 = € 6.00*). Quindi occorre conteggiare un prelievo totale di *€ 46.00* , per cui il saldo finale di *€ 54.00*.
  d) in corrispondenza a c), l'oggetto *ContoVincolato* del cliente "Lupo", di saldo iniziale zero, è soggetto ad un versamento complessivo di *€ 40.00* in 6 transazioni, di cui l'ultimo prelievo produce l'addebito della penale di *€ 20.00*; quindi in sostanza si ha un versamento complessivo di *€ 20.00*, a cui vanno aggiunti gli interessi del *3%* , pari a *€ 20.00 * 0.03 = € 0.6*, e quindi il saldo finale di *€ 20.6*.
- Si noti che il metodo `equals` ridefinito per la classe `Conto` considera equivalenti anche due oggetti `ContoCorrente` e `ContoVincolato` che abbiano il medesimo saldo, in accordo alle politiche della banca. Infatti `B` **`instanceof`** `A` ritorna `true` se è possibile effettuare il cast di `B` in `A`, il che è vero se `B` è una classe estesa di `A`.

```
// output

Banca [
 totClienti = 4;
Cliente [
 nome = Romolo;
 numero = 101;
 pin = 1234;
Conto [
 saldo = 70.0
]
]
Cliente [
 nome = Remo;
 numero = 102;
 pin = 2341;
ContoCorrente [
 totTransazioni = 0;

ContoCorrente [
 saldo = 54.0
]
]
]
Cliente [
 nome = Lupa;
 numero = 103;
 pin = 3412;
LibrettoDiRisparmio [
 tassoInteresse = 1.5;

LibrettoDiRisparmio [
 saldo = 30.45
]
]
]
Cliente [
 nome = Lupo;
 numero = 104;
 pin = 4123;
ContoVincolato [
 periodoMaturazione = 11;

ContoVincolato [
 tassoInteresse = 3.0;

ContoVincolato [
 saldo = 20.6
]
]
]
]
]
true
```

Esempio di utilizzo di javadoc (the Java API Documentation Generator)

Lo strumento *javadoc* genera automaticamente, in formato html, la documentazione di una classe a partire dal suo codice, con la medesima impaginazione della documentazione ufficiale Java.

```
javadoc *.java –d documentazione
```

[7] Per il metodo **main** (e d'ora in avanti) valgono le medesime considerazioni di leggibilità della precedente esercitazione.

produce una documentazione completa di tutti i metodi pubblici, ma (ovviamente) priva di commenti e spiegazioni, accessibile con un browser a partire dal file index.html nella cartella `documentazione`.
Inserire le seguenti righe di commento nel codice sorgente della classe *Conto*, e rigenerare la documentazione.

DOVE	COSA

all' inizio del file Conto.java	`/**`
	` * Un oggetto di classe <code> Conto </code> rappresenta un conto corrente`
	` * ed &eacute; costituito da un <em> numero di conto </em>, che lo identifica`
	` * univocamente, ed un <em>saldo</em> che specifica la quantit&aacute; di denaro`
	` * presente sul conto.`
	` * esempio:`
	` * <pre>`
	` * Conto mioConto = new Conto(1000.0);`
	` * mioConto.prelievo(100.0);`
	` * </pre>`
	` *`
	` * @author Gianluca Dini`
	` * @version 1.0`
	` * @see Banca`
	` */`

prima di
`public Conto () {`
	`/**`
	` * Crea un conto ed imposta il saldo a zero`
	` */`

prima di
`public Conto (double saldoIniziale) {`
	`/**`
	` * Crea un conto ed imposta il saldo a <code> saldoIniziale </code>`
	` * @param saldoIniziale valore iniziale del saldo`
	` */`

prima di
`public double ritornaSaldo () {`
	`/**`
	` * Ritorna il saldo del conto`
	` * @return il saldo`
	` */`

Per far comparire anche le informazioni su autore e versione (laddove presenti) digitare:

```
javadoc *.java –d documentazione –author –version
```

Notare che:
- gli asterischi sono automaticamente tolti dai commenti
- tutto ciò che é tra `<pre>` e `</pre>` viene "preformattato" cioè mantenuta l'indentazione (spaziature) con un font *a larghezza fissa* (in cui ogni carattere occupa il medesimo numero di pixel, del tipo *`Courier`*) propri del codice sorgente. Lo stesso effetto si ottiene con i tag `<code> ... </code>` quest' ultimo adoperato per piccoli frammenti di codice.
Il tag `<em>` sta per testo *emphasized* e in genere il browser lo intrepreta come corsivo. Infine `<b> ... </b>` si può adoperare per il grassetto (bold), e `<BR>` per andare accapo.

Per ulteriori dettagli sui tag del linguaggio HTML : http://www.w3.org/TR/html401/index/elements.html

Per ulteriori informazioni su Javadoc:
http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/javadoc.html

ESERCITAZIONE TIGA: File, compareTo, package, jar

1) In aggiunta a quanto specificato nella esercitazione su *ereditarietà*, sulla *banca* possono essere fatte (almeno) le operazioni:

- `public void salva(String fnome)` che salva questa banca sul file il cui nome è specificato dal parametro `fnome`. Supponendo che l'operazione sia eseguita quotidianamente, il nome del file deve essere strutturato come segue: "`banca.`" + *<data attuale>* + "`.dat`", dove *<data attuale>* indica la data in cui viene eseguita l'operazione, nel formato `giorno-mese-anno`[8].
- `public void ripristina(String fnome)` che legge questa banca dal file il cui nome è specificato da `fnome`, ricostruendo lo stato dell'oggetto al momento del salvataggio.
- `public void elaboraESalvaStatistiche(String fnome)` che *aggiunge* ad un file di testo, il cui nome è specificato dal parametro `fnome`, le seguenti informazioni:

```
n. di clienti della banca <tabulazione> importo complessivo dei saldi depositati <invio>
```

- `public void caricaEdElaboraStatistiche(String fnome)` che legge dal file di testo, il cui nome è specificato dal parametro `fnome`, le varie registrazioni precedentemente eseguite mediante `salvaStatistiche` (oppure inserite manualmente mediante un text editor, o inserite da altre applicazioni che scrivono nel file rispettando il formato suddetto), e visualizza sullo schermo i due valori medi (del n. di clienti e dell'importo complessivo dei saldi) sul totale di registrazioni.
- `public void ordina()` che riordina l'array di clienti adoperando il metodo `sort` della classe `Arrays`. A tale scopo, si implementi il metodo `compareTo` dell'interfaccia `Comparable` sia in `Cliente`[9] che in `Conto`, basando il confronto esclusivamente sul valore del `saldo`. Si faccia l'ipotesi semplificativa che l'array non contenga riferimenti a `null`[10].

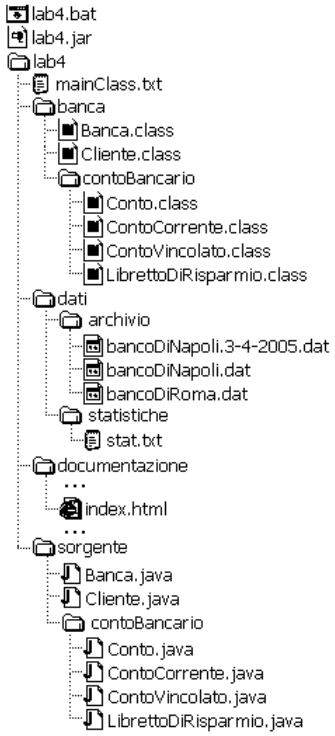


Fig.1 riorganizzazione dei file

2) Riorganizzare i file dell'intera applicazione banca come in Fig.1. Creare una cartella *lab4*. Il package *lab4.banca* (cartella *banca*) contiene le classi *Banca* e *Cliente*, ed il package *lab4.banca.contoBancario* (cartella *banca/contoBancario*) contiene i diversi tipi di conto. I file sorgenti sono invece nella cartella *sorgente* con la medesima struttura vista per i corrispondenti package. La cartella *dati* contiene i file prodotti dai metodi `salva` (in una sottocartella *archivio*) ed `elaboraEsalvaStatistiche` (in una sottocartella *statistiche*). Infine, la cartella *documentazione* contiene le pagine web prodotte da javadoc.

3) Creare un unico file *lab4.jar*, contentente solo i file .class, mediante il Java Archiver (JAR), con il seguente comando (da Shell):

```
jar cmf lab4/mainClass.txt lab4.jar lab4/banca
```

dove `mainClass.txt` é un file di testo che indica quale classe possiede il metodo main, nel seguente formato[11] :

```
Main-Class: lab4.banca.Banca ↵
```

4) Portare il file jar in un altro path, ed eseguirlo:

```
java -jar lab4.jar <eventuali parametri>
```

Funziona ancora ? Adesso, dove vengono salvati e letti i file dati ?

5) Inserire tutti i comandi di compilazione ed esecuzione, produzione ed esecuzione del *.jar*, etc.., in un file di testo di tipo *batch "lab4.bat"*, ossia uno script contenente una sequenza di comandi per la Shell, invocabile da riga di comando come *"lab4"*.

---

[8] Si consigliano le classi `Calendar` e `GregorianCalendar` del pacchetto `java.util`. Si noti che la classe `Calendar` è astratta.

[9] Il metodo `compareTo` di `cliente` si limita ad invocare il metodo `compareTo` di `conto`, dopo aver eseguito un cast.

[10] Altrimenti il metodo `Arrays.sort` si bloccherebbe generando l'eccezione `lava.lang.NullPointerException`. Questo problema si potrebbe risolvere, ad esempio, adoperando un oggetto *Cliente dummy* ("fantoccio") al posto del *null*, definito come una costante privata e statica della classe `Banca`:

```
private static final Cliente DUMMY = new Cliente ("", 0, 0, new Conto());
```

e rivedendo il costruttore e tutti i metodi di `Banca` in modo che i riferimenti a `null` nell'array `elencoClienti` siano convertiti in riferimenti all'oggetto `DUMMY`, e cambiando i test "`== null`" in "`== DUMMY`" (non occorre usare `equals`).

[11] Assicurarsi di battere anche il ritorno carrello (↵), in assenza del quale, nei sistemi Windows©, il JAR non inserisce nel file manifesto le informazioni sulla Main-Class, per cui l'interprete java produce a run-time l'errore `Failed to load Main-Class manifest attribute`.

ESERCITAZIONE TIGA: File, compareTo, package, jar – SOLUZIONE PROPOSTA

Progettazione:

Per memorizzare lo stato di un oggetto su di un altro oggetto persistente (file), é necessario decidere un formato di memorizzazione degli attributi in una struttura sequenziale, adatta ad essere inviata ad uno stream associato ad un file. Questa operazione non é banale, se si pensa alle numerose relazioni che possono esistere tra gli oggetti (ereditarietà, composizione,...). Il procedimento inverso, la ricostruzione della struttura originaria di un oggetto, è altrettanto complesso.

In Java questo problema é stato risolto in modo generale, introducendo diversi tipi di stream e l'interfaccia di serializzabilità; dichiarando che una classe implementa l'interfaccia *Serializable*, si indica che un relativo oggetto è serializzabile, quindi é possibile adoperare il metodo *writeObject* (risp. *readObject*) il quale trasmette (riceve) tale oggetto a (da) un *Object stream*. Questo object stream, a sua volta, può essere associato ad un *file stream*, ma anche ad un *network socket stream* (per permettere la ricostruzione di un oggetto su un altro host o in un altro processo). Tutte le sottoclassi di una classe serializzabile sono automaticamente serializzabili, mentre le classi *contenute* e le superclassi non lo sono.

Per sapere se una classe è serializzabile, avendo a disposizione il file .class, basta digitare ad esempio[12]:

```
serialver Banca
```

La *readObject* implica la costruzione dell'oggetto (invocare i costruttori dei suoi componenti) a partire dall' object stream; in casi eccezionali, possono mancare delle informazioni nel flusso di provenienza. Altre situazioni eccezionali possono avvenire nella gestione dei file (es. si raggiunge precocemente la fine del file), per cui questi metodi richiedono obbligatoriamente la gestione di alcune eccezioni (`IOException, java.lang.ClassNotFoundException`) che, per semplicità, verranno rilanciate dai metodi (dichiarate *throws* nella intestazione) ma non catturate (*catch*), per cui verranno propagate alla classe che adopera il metodo, e via via fino al *main*.

In UML (Fig.1) l'interfaccia si può rappresentare separatamente dalle entità che la implementano, come una classe etichettata con lo stereotipo «interface». La relazione tra le classi che implementano e l'interfaccia è la *realizzazione*, rappresentata in modo simile all'*ereditarietà* ma tratteggiata (in effetti una interfaccia, al pari di una superclasse, è una parte dei servizi di una classe); *Banca* invece ha una relazione di *dipendenza* ossia una relazione in cui la modifica di una classe (interfaccia, nella fattispecie) necessita la modifica di un'altra (la modifica del codice sorgente).

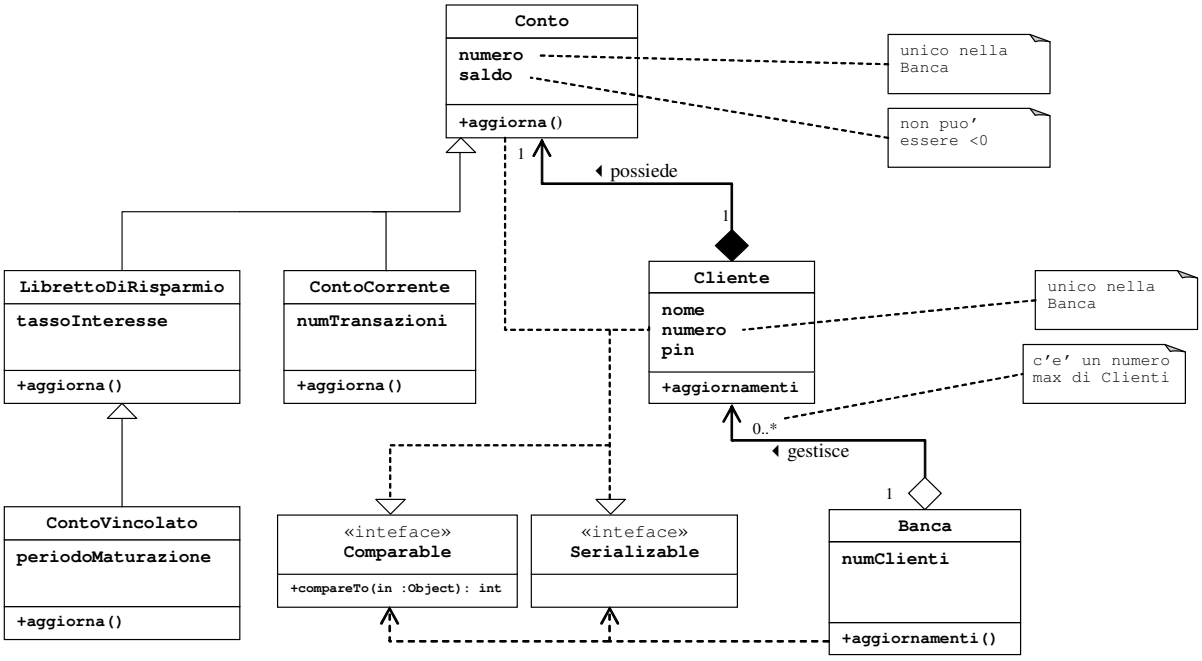


Fig.1 – Diagramma delle classi (semplificato) del sistema *Banca*.

L'interfaccia *Comparable* serve a definire, tramite il metodo *compareTo*, un ordinamento naturale tra gli oggetti delle classi `Cliente` e `Conto`.

La serializzazione produce un file *binario*, non intellegibile. La scrittura (risp. lettura) su file di *testo* viene realizzata adoperando la classe *FileWriter* (*FileReader*) che associa uno stream di caratteri (indipendente dal sistema operativo sottostante) ad un file. A sua volta, lo stream di caratteri viene scritto (risp. letto) a partire dagli oggetti, mediante il metodo *println* (*readLine*) della classe *PrintWriter* (*BufferedRader*), che ha come argomento (risultato) una stringa, o un oggetto reso tale, in modo implicito, da *toString*.

Nel caso particolare della lettura, una riga di testo va analizzata (*parsing*) e spezzata in sottostringhe; a tale scopo, la classe *StringTokenizer* permette la suddivisione di una stringa in token (un token è la rappresentazione di un simbolo) specificando il

[12] *Serialver* è un'altra applicazione della directiry bin. Nella fattispecie occorrerà specificare anche il package, quindi `serialver lab4.banca.Banca`, che risulta non serializzabile. Invece la classe `lab4.banca.Cliente` è serializzabile.

delimitatore (tabulazione nella fattispecie), mentre la classe Wrapper Double, con il metodo *parseDouble*, permette l'analisi e la conversione di stringhe (token) nei rispettivo tipo elementare.

Ulteriori dettagli sugli strumenti adoperati:

In Windows XP ©, nell'inserimento del CLASSPATH, fare attenzione a NON inserire spazi tra un path e l'altro, a NON mettere alcun ";" in fondo, ed a chiudere e riaprire le Shell eventualmente già aperte perchè la modifica abbia effetto. Esempio di CLASSPATH:

```
C:\Program Files\j2sdk_nb\j2sdk1.4.2\bin\java;D:\_TIGA\java\labs
```

Codifica in linguaggio Java (modifiche rispetto al laboratorio su *ereditarietà*):

```
// LibrettoDiRisparmio.java
// ContoVincolato.java
// ContoCorrente.java
(aggiungere in tutte le tre classi, all'inizio, il seguente rigo)
package lab4.banca.contoBancario;

// Conto.java
(aggiungere, all'inizio, e modificare l'intestazione della classe come segue)

package lab4.banca.contoBancario;

import java.io.*;
/*
...
*/
public class Conto implements Serializable, Comparable {

    // ...
    (aggiungere, es. come ultimo metodo)

    public int compareTo(Object obj) {
        Conto altro = (Conto)obj;             // possibile ClassCastException
        if ( saldo < altro.saldo )
            return -1;
        if ( saldo > altro.saldo )
            return 1;
        return 0;
    }

// Cliente.java
(aggiungere, all'inizio, e modificare l'intestazione della classe come segue)

package lab4.banca;

import java.io.*;
import lab4.banca.contoBancario.*;

public class Cliente implements Serializable, Comparable {

     // ...
    (aggiungere, es. come ultimo metodo)

    public int compareTo(Object o) {
        Cliente altro = (Cliente)o;             // possibile ClassCastException
        return conto.compareTo(altro.conto);
    }

// Banca.java
(aggiungere, all'inizio, i seguenti 4 righi)

package lab4.banca;

import java.io.*;
import java.util.*;
import java.util.Arrays;
import lab4.banca.contoBancario.*;

(aggiungere, es. prima delle definizioni private, la parte che segue)

     public void salva (String fnome) throws IOException  {
        FileOutputStream fout = null;
        ObjectOutputStream oout = null;

        fout = new FileOutputStream(fnome);
         oout = new ObjectOutputStream(fout);
          oout.writeObject(elencoClienti);
          oout.writeInt(totClienti);
         oout.close();
        fout.close();
```

```
    }

    public void ripristina (String fnome)   throws IOException,
                                            java.lang.ClassNotFoundException {
        FileInputStream fin = null;
        ObjectInputStream oin = null;

        fin = new FileInputStream(fnome);
         oin = new ObjectInputStream(fin);
          elencoClienti = (Cliente[])oin.readObject();
          totClienti = oin.readInt();
         oin.close();
        fin.close();
    }

     public void elaboraESalvaStatistiche (String fnome) throws IOException {
        double saldoTot = 0;
        if (totClienti != 0) {
            for (int i = 0; i < elencoClienti.length; i++) {
                if (elencoClienti[i] != null)
                    saldoTot += elencoClienti[i].ritornaSaldo();
            }
        }

        FileWriter fout = null;
        PrintWriter dout = null;

        fout = new FileWriter(fnome, true);
         dout = new PrintWriter(fout);
          dout.println(totClienti + "\t" + saldoTot);
         dout.close();
        fout.close();
    }

    public void caricaEdElaboraStatistiche (String fnome) throws IOException {
        double numClientiMedio = 0.0, saldoTotMedio = 0.0;
        FileReader fin = null;
        BufferedReader din = null;
        int numLine = 0;
        String linea = null;
        StringTokenizer parola = null;

        fin = new FileReader(fnome);
         din = new BufferedReader(fin);
          linea = din.readLine();
          while (linea != null) {
                parola = new StringTokenizer(linea,"\t");
                numClientiMedio += Double.parseDouble(parola.nextToken());
                saldoTotMedio += Double.parseDouble(parola.nextToken());
                numLine++;
                linea = din.readLine();
          }
         din.close();
        fin.close();

        if (numLine != 0) {
            numClientiMedio /=numLine;
            saldoTotMedio /=numLine;
        }

        System.out.println("Su " + numLine + " registrazioni, " +
                           "il numero di Clienti medio e' " +
                           Math.round(numClientiMedio*100)/100.0 +
                           ", ed il saldo totale medio " +
                           Math.round(saldoTotMedio*100)/100.0 );
    }

    public void ordina() {
        Arrays.sort(elencoClienti);                             // presuppone nessun null
    }


(riscrivere il metodo main come segue)

    public static void main(String[] args) throws IOException,
                                           java.lang.ClassNotFoundException {
        Banca bancoDiRoma = new Banca(4);

        bancoDiRoma.registra (new Cliente ("Romolo", 101, 1234, new Conto(100.00) ));
        bancoDiRoma.registra (new Cliente ("Remo"  , 102, 2341, new ContoCorrente(100.00) ));
        bancoDiRoma.registra (new Cliente ("Lupa"  , 103, 3412, new LibrettoDiRisparmio(1.5)));
        bancoDiRoma.registra (new Cliente ("Lupo"  , 104, 4123, new ContoVincolato(3.0, 12)));
```

```
        ( bancoDiRoma.trova("Romolo") ).trasferisci( bancoDiRoma.trova("Lupa"), 40.00);
        ( bancoDiRoma.trova("Remo") ).trasferisci( bancoDiRoma.trova("Lupo"), 30.00);

        bancoDiRoma.aggiornamenti();
        System.out.println(bancoDiRoma.toString());
        bancoDiRoma.ordina();
        System.out.println(bancoDiRoma.toString());

        final String percorsoDati = args[0] + args[1];
        final String percorsoStat = args[0] + args[2];

        bancoDiRoma.salva(percorsoDati + "bancoDiRoma.dat");

        Banca bancoDiNapoli = new Banca();
        bancoDiNapoli.ripristina(percorsoDati + "bancoDiRoma.dat");
        bancoDiNapoli.salva(percorsoDati + "bancoDiNapoli.dat");

        GregorianCalendar cal = new GregorianCalendar();
        int anno = cal.get(Calendar.YEAR);
        int mese = cal.get(Calendar.MONTH)+1;
        int giorno = cal.get(Calendar.DAY_OF_MONTH);

        String dataAttuale = giorno +  "-" + mese + "-" + anno;

        bancoDiNapoli.salva(percorsoDati + "bancoDiNapoli." + dataAttuale + ".dat");

        bancoDiNapoli.elaboraESalvaStatistiche(percorsoStat + "stat.txt");
        bancoDiNapoli.caricaEdElaboraStatistiche(percorsoStat + "stat.txt");

    } // main
```

Testing del programma:

Per testare il codice, poniamo nel medesimo percorso della cartella *lab4* il seguente script *lab4.bat*:

```
@echo off
set CLASSPATH=
setlocal
set CURRENT_DIR=%cd%
javac lab4/sorgente/contoBancario/*.java -d ./
javac -classpath "%CURRENT_DIR%" lab4/sorgente/*.java -d ./
javadoc -d lab4/documentazione lab4/sorgente/*.java lab4/sorgente/contoBancario/*.java
java -classpath "%CURRENT_DIR%" lab4.banca.Banca "%CURRENT_DIR%"/lab4/ dati/archivio/ dati/statistiche/
jar cmf lab4/mainClass.txt lab4.jar lab4/banca
java -jar lab4.jar "%CURRENT_DIR%"/lab4/ dati/archivio/ dati/statistiche/
fc lab4\dati\archivio\bancoDiNapoli.dat lab4\dati\archivio\bancoDiRoma.dat
endlocal
@echo on
```

Digitando "*lab4*" da Shell vengono eseguite automaticamente le seguenti procedure:

- compilazione di tutti i conti bancari e produzione dei file *.class* a partire dalla cartella corrente[13]
- compilazione delle altre classi e produzione dei file *.class* a partire dalla cartella corrente[13]
- generazione della documentazione nella cartella *documentazione*
- esecuzione dell'applicazione, con indicazione del classpath, e passaggio di tre stringhe come parametri
- creazione del jar
- esecuzione del jar
- confronto di due archivi prodotti

Nello script, si possono automaticamente invocare sia strumenti della Java Platform, sia istruzioni della Shell. Nell'ultimo rigo, si verifica se due file contenenti due banche archiviate siano identici o meno; Per ulteriori dettagli, digitare `help` (Win) o `man` (Linux), brevemente:

- per confrontare due file:

`comp` nomefile1 nomefile2 (Windows) oppure
`fc` nomefile1 nomefile2 (Windows, comando più dettagliato)
`diff` nomefile1 nomefile2 (Unix)

- per ottenere le dimensioni di file e cartelle (in bytes) da Shell:

`dir` nomefile o nomecartella /s (Windows)
`du` nomefile o nomecartella -s –b (Unix)

Quest'ultimo comando può servire per confrontare il jar e l'insieme dei file .class. La Shell offre anche costrutti tipo IF per modificare il flusso di controllo degli script in base a parametri, variabili ambiente, risultati dei comandi precedenti.

Eseguendo il jar in altro percorso, e dando come parametri delle stringhe vuote, i file verranno generati localmente:

```
java -jar lab4.jar "" "" ""
```

[13] In tal caso *javac* produce automaticamente le sottocartelle dei package, se queste non esistono.

Dal punto di vista tecnologico, un .jar è una compressione di classi e package in un unico file .zip, che viene automaticamente decompresso con l'opzione "-jar" prima di essere eseguito dalla Java Virtual Machine. Provare ad aprirlo con Winzip per vederne il contenuto.

```
// output
```

In uscita, vengono visualizzati:
- messaggi relativi al javadoc (si raccomanda di sfogliare la documentazione, ponendo attenzione alla struttura dei package nel frame di sinistra della pagina html)
- un output simile al laboratorio su *ereditarietà*, tranne per il fatto che le classi appaiono con il loro nome completo del package (es. `lab4.banca.Banca`)
- un output in cui le classi sono ordinate per importo (dal più piccolo)
- il seguente messaggio (dipendente dai valori inseriti nel file stat.txt):

```
Su 21 registrazioni, il numero di Clienti medio e' 4.33, ed il saldo totale medio 484.58
```

- di nuovo l'elenco dei clienti, relativo all'esecuzione del jar
- un messaggio di confronto dei due file:

```
Confronto in corso dei file LAB4\DATI\ARCHIVIO\bancoDiNapoli.dat e LAB4\DATI\ARCHIVIO\BANCODIROMA.DAT

FC: nessuna differenza riscontrata
```

ESERCITAZIONE TIGA: NetBeans IDE, Eccezioni, Socket, rete.

Un' azienda che opera nel settore finanziario necessita di eseguire, occasionalmente ma con certezza di integrità, il backup di una quantità limitata di dati. A tale scopo, viene deciso di adoperare tre host, collocati ad almeno 300 km di distanza l'uno dall'altro. Uno degli host é locale, ed i dati vengono inviati attraverso la rete Internet adoperando il protocollo TCP, nella seguente modalità (Fig.1).

L' host che possiede i dati in un file (*Host0*) li invia ad un altro host (*Host1*); questi a sua volta li memorizza in un file, li rilegge e li invia al terzo host (*Host2*), il quale esegue la medesima procedura rinviando i dati all'host di partenza (Host0); infine questi esegue un confronto per uguaglianza dei dati. In tal modo, viene rilevata qualsiasi alterazione sia sul trasporto che sulla memorizzazione.

Sviluppare un' unica applicazione in Java che, collocata sui diversi host ed eseguibile in due diverse modalità, realizzi il servizio descritto, provvedendo a catturare e gestire le seguenti eccezioni:

`IOException`
`ClassNotFoundException`

visualizzando un messaggio all' utente e garantendo la chiusura dei flussi prima della terminazione.

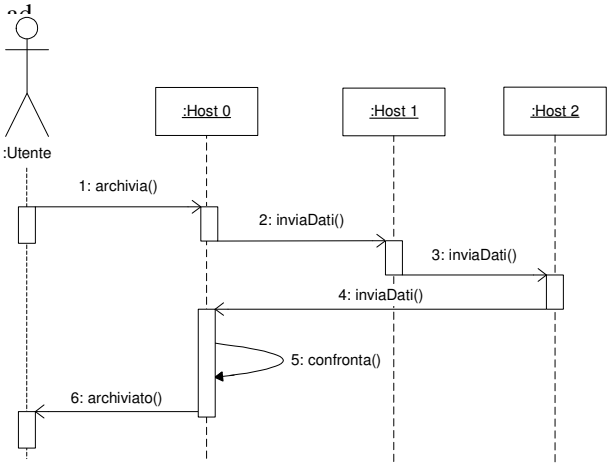


Fig.1 – Diagramma di sequenza del protocollo di backup.

Suggerimenti:

• Supponendo per semplicità che i dati siano costituiti da una sola stringa, si implementi una classe `Host` dal campo stringa e dai seguenti metodi (per gli ultimi due, riusare il codice dei metodi `salva` e `ripristina` del laboratorio su *file*) :

```
public void          inviaDati(String indirizzoIP, int porta)
public String        riceviDati(int porta)
public void          salvaSuFile (String fnome)
public void          leggiDaFile (String fnome)
```

• Si implementi poi una classe `SecureBackup` contenente solo il main, la quale riceve tramite `args` il parametro *modalità* , un intero che indica se l'host deve eseguire le operazioni di *Host0* (cioè creare il file con l'oggetto stringa serializzato, rileggerlo, inviare la stringa e poi aspettare che ritorni dall'ultimo host) oppure di *Host1* (cioè ricevere la stringa, salvarla su file, rileggerla ed inoltrarla all'host successivo). Gli altri parametri sono: 1) il nome del file da creare; 2) l'indirizzo IP dell'host di destinazione; 3) la porta locale di ascolto; 4) la porta remota di ascolto per il server successivo; 5) la stringa da archiviare (per l'*Host0*). Ecco alcuni esempi di esecuzione dell'applicazione:

a) esecuzione su diversi host

SU QUALE HOST	COMANDO DA SHELL	DESCRIZIONE
Host0	`java SecureBackup 0 MioFile.dat 131.121.191.162 8080 8080 CIAO`	Ogni host ha l'indirizzo IP dell' host di destinazione ed ascolta sulla porta 8080.
Host1	`java SecureBackup 1 MioFile.dat 131.124.341.111 8080 8080`	
Hostk	`java SecureBackup 1 MioFile.dat 132.114.291.213 8080 8080`	

b) esecuzione in locale

SU QUALE HOST	COMANDO DA SHELL	DESCRIZIONE
Host0	`java SecureBackup 0 MioFile0.dat 132.114.291.213 8082 8080 CIAO`	L'unico Host può essere riferito anche con l'indirizzo IP di loopback 127.0.0.1 oppure con "". Le porte di ascolto devono susseguirsi per Host adiacenti.
Host1	`java SecureBackup 1 MioFile1.dat 132.114.291.213 8080 8081`	
Hostk	`java SecureBackup 1 MioFile2.dat 132.114.291.213 8081 8082`	

N.B.: *Host0* deve essere avviato per ultimo, dopo che gli altri host sono tutti in "ascolto".

Si ricorda, in ambienti Windows©, di digitare `set CLASSPATH=` all' apertura della Shell.

Progettazione:

In Fig.1 mostriamo la distribuzione dei componenti dell'applicazione Java `SecureBackup`. Ogni nodo, rappresentato da una scatola, corrisponde ad un host, ossia un oggetto fisico presente nell'ambiente di esecuzione, dotato di memoria e capacità di elaborazione, sul quale possono essere ubicati dei componenti e degli oggetti. L'interfaccia di comunicazione con l'host viene schematizzata come la coppia <indirizzoIP:numero di porta>.

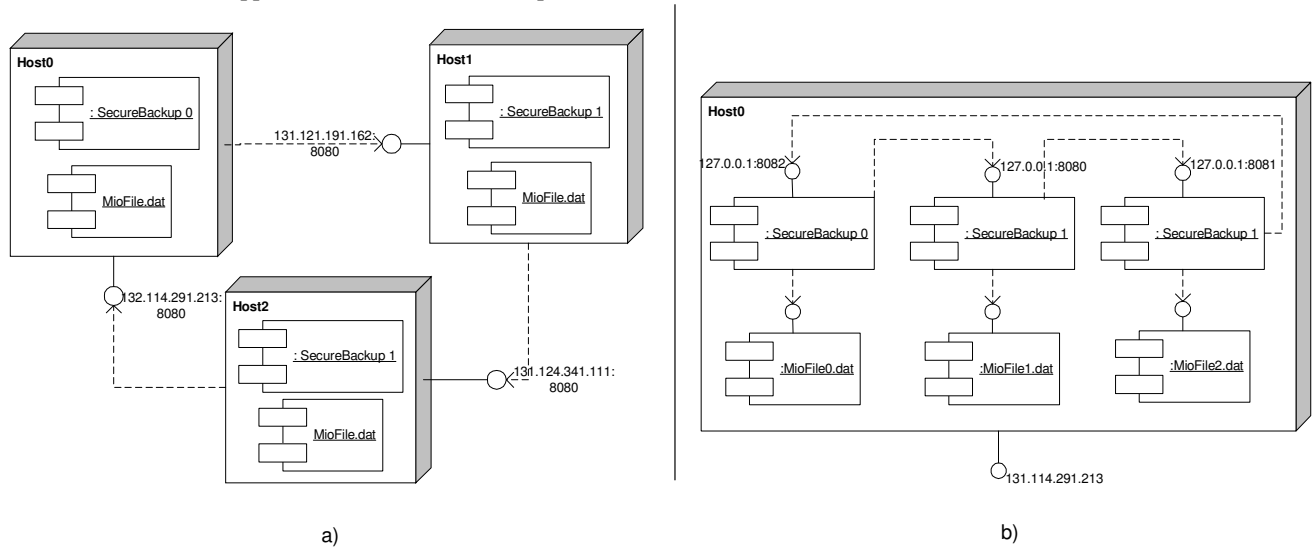


Fig.1 – Diagramma di distribuzione dell'applicazione SecureBackup, a) con più host e b) con un solo host.

Tutti i computer connessi alla rete Internet (host) possiedono un loro indirizzo unico, detto indirizzo IP (Internet Protocol) che si compone di quattro cifre decimali, ognuna delle quali è compresa tra 0 e 255. Esiste un servizio, detto DNS (Domain Name System), che permette di riferire un indirizzo IP in modo mnemonicamente più significativo (es. http://info.iet.unipi.it). Ci sono indirizzi IP *pubblici*, univoci in tutto il mondo ed accessibili da qualsiasi punto di Internet, ed indirizzi *privati*, visibili sono in una rete interna. Tra gli indirizzi "speciali", quello di loopback, che inizia con 127, è un indirizzo privato che ciascun host possiede per riferire sè stesso.

Affinché un computer possa comunicare con un altro mediante il protocollo TCP (Transmission Control Protocol) è necessario che su entrambi i computer sia disponibile un *socket* (astrazione per indicare una "presa di attacco"), un punto finale di comunicazione tra due host, una sorta di handle (gestore) TCP. L'utilizzo di un socket comporta innanzitutto l'apertura di una *porta* TCP, ulteriore astrazione che denota un punto "logico" di connessione, locale all'host, al fine di permettere al TCP di gestire più flussi di dati contemporaneamente con diverse applicazioni. Poi, la messa in ascolto di un host (che per questo viene detto "server") che attende che un altro/i host (detti "client") si colleghino con una chiamata, conoscendo la coppia <indirizzoIP:numero di porta>. In Java è la classe `ServerSocket` che si occupa di gestire l'"ascolto" passivo del server su una porta, mediante il metodo `accept()` che produce un blocco del flusso di controllo, in attesa di richieste di connessione. Una volta avvenuta la richiesta, viene creato un `socket` ed un `InputStream`, al quale si può associare ad esempio un `ObjectInputStream` , da gestire in modo identico alla lettura di file. Il Client, da parte sua, per inviare richieste crea un `socket`, al quale viene associato un `OutputStream`, gestibile in modo identico alla scrittura di file.

La gestione delle **eccezioni** produce la visualizzazione di un messaggio specifico delle istruzioni che l'hanno prodotta. Si osservi che, per evitare che le stringhe di messaggi corrompano la struttura visiva del codice di controllo, esse sono state raccolte a parte, nell' area di memoria statica. Si ricorda che il blocco `finally` non può essere evitato dal controllo di flusso dell'aplicazione: le istruzioni `break, continue, return` all' interno del blocco `try` o `catch` verranno eseguite solo dopo l'esecuzione del blocco `finally`.

Codifica in linguaggio Java:

```
// Host.java
import java.io.*;
import java.net.*;
import java.util.*;

public class Host {
    private String dati;

    public Host(String dato) {
        dati = dato;
    }
    public String ritornaDati() {
        return dati;
    }
```

```
public void aggiornaDati(String dato) {
    dati = dato;
}
public void inviaDati(String indirizzoIP, int porta) {
    InetAddress         ind = null;
    Socket              socket = null;
    ObjectOutputStream  oout = null;

    try {
        ind = InetAddress.getByName(indirizzoIP);
        socket = new Socket(ind, porta);
        oout = new ObjectOutputStream (socket.getOutputStream());
        oout.writeObject(dati);
    }
    catch (IOException e) {
        System.err.println(MSG_ERR9 + e);
    }
    finally {
        try {
            oout.close();
            socket.close();
        }
        catch (IOException e) {
            System.err.println(MSG_ERR10 + e);
        }
    }
}
public String riceviDati(int porta) {
    String              result = null;
    ServerSocket        servs = null;
    Socket              socket = null;
    ObjectInputStream   oin = null;

    try {
        servs = new ServerSocket(porta);
        socket = servs.accept();
        oin = new ObjectInputStream( socket.getInputStream());
        result = (String)oin.readObject();
    }
    catch (IOException e) {
        System.err.println(MSG_ERR6 + e);
    }
    catch (ClassNotFoundException e) {
        System.err.println(MSG_ERR7 + e);
        e.printStackTrace();
    }
    finally {
        try {
            oin.close();
            servs.close();
            socket.close();
        }
        catch (IOException e) {
            System.err.println(MSG_ERR8 + e);
        }
    }
    return result;
}
public void salvaSuFile (String fnome) {
    FileOutputStream fout = null;
    ObjectOutputStream oout = null;
    try {
         fout = new FileOutputStream(fnome);
         oout = new ObjectOutputStream(fout);
         oout.writeObject(dati);
    }
    catch (IOException e) {
        System.err.println (MSG_ERR1 + e);
    }
    finally {
        try {
            oout.close();
            fout.close();
        }
        catch (IOException e) {
            System.err.println (MSG_ERR2 + e);
        }
    }
}
public void leggiDaFile (String fnome) {
    FileInputStream fin = null;
    ObjectInputStream oin = null;
```

```
        try {
            fin = new FileInputStream(fnome);
            oin = new ObjectInputStream(fin);
            dati = (String)oin.readObject();
        }
        catch (IOException e) {
            System.err.println(MSG_ERR3 + e);
        }
        catch (ClassNotFoundException e) {
            System.err.println(MSG_ERR5 + e);
            e.printStackTrace();
        }

        finally {
            try {
                oin.close();
                fin.close();
            }
            catch (IOException e) {
                System.err.println (MSG_ERR4 + e);
            }
        }
    }
    private static final String WORD1 = "Impossibile ";
    private static final String WORD2 = ": ";

    private static final String MSG_ERR1  = WORD1 + "aprire o scrivere i flussi di uscita" + WORD2;
    private static final String MSG_ERR2  = WORD1 + "chiudere i flussi di uscita"  + WORD2;
    private static final String MSG_ERR3  = WORD1 + "aprire o leggere i flussi di ingresso"  + WORD2;
    private static final String MSG_ERR4  = WORD1 + "chiudere i flussi di uscita"  + WORD2;
    private static final String MSG_ERR5  = WORD1 + "ricostruire l'oggetto dal flusso"  + WORD2;
    private static final String MSG_ERR6  = WORD1 + "aprire o leggere il socket"  + WORD2;
    private static final String MSG_ERR7  = WORD1 + "ricostruire l'oggetto dal socket"  + WORD2;
    private static final String MSG_ERR8  = WORD1 + "chiudere il flusso di ingresso o il socket"  + WORD2;
    private static final String MSG_ERR9  = WORD1 + "aprire o scrivere i flussi di uscita o i socket" +
WORD2;
    private static final String MSG_ERR10 = WORD1 + "chiudere il flusso di uscita o il socket" + WORD2;
}

// SecureBackup.java
import java.io.*;
public class SecureBackup {
    public static void main(String args[]) {
        if (args.length < 5) {
            System.err.println(MSG_USR1);
            System.exit(1);
        }
        final int     modo        = Integer.parseInt(args[0]);
        final String  nomeFile    = args[1];
        final String  indirizzoIP = args[2];
        final int     portaSrc    = Integer.parseInt(args[3]);
        final int     portaDst    = Integer.parseInt(args[4]);
        final String  dato        = (modo==0) ? args[5]: null;

        Host mioHost = new Host(dato);
        String datoRicevuto = null;
        switch (modo) {
            case 0: mioHost.salvaSuFile(nomeFile);
                    mioHost.leggiDaFile(nomeFile);
                    mioHost.inviaDati(indirizzoIP, portaDst);
                    datoRicevuto = mioHost.riceviDati (portaSrc);
                    if ( datoRicevuto.equals(mioHost.ritornaDati()) )
                        System.out.println(MSG_USR2);
                    else
                        System.out.println(MSG_USR3);
                    break;
            case 1: datoRicevuto = mioHost.riceviDati (portaSrc);
                    mioHost.aggiornaDati(datoRicevuto);
                    mioHost.salvaSuFile(nomeFile);
                    mioHost.leggiDaFile(nomeFile);
                    mioHost.inviaDati(indirizzoIP, portaDst);
                    break;
        }
    }
    private static final String MSG_USR1 = "java SecureBackup <modo> <nomefile>" +
                         " <ind. IP dest> <porta ascolto> <porta invio> <stringa" +
                         "messaggio>\n dove <modo> e' 0 (host di partenza) oppure" +
                         "1 (altri host)";
    private static final String MSG_USR2 = "Dato archiviato correttamente";
    private static final String MSG_USR3 = "Errori nell' archiviazione";
}
```

Testing del programma:

- Eseguire l'applicazione come indicato nelle tabelle a) o b) della traccia. Se tutto funziona bene, compare sull' *Host0* la stringa "Dato archiviato correttamente". Si possono anche confrontare con il comando *fc* i MioFile.dat che vengono prodotti.

- Per conoscere l'indirizzo IP ed il nome simbolico di un host locale, digitare `ipconfig /all` ; per conoscere il *nome a dominio* (nome simbolico completo di tutti i nomi, ai vari livelli di dominio, separati da punti) dato l'indirizzo IP di un host, digitare ad es. `nslookup 131.114.9.184` (compare `info.iet.unipi.it`). Vale anche il viceversa[14].

- Per verificare se un host remoto è *attivo* (= risponde al protocollo IP), e valutarne i tempi di risposta, digitare `ping`[15] seguito dal nome simbolico dell'host o dall'indirizzo IP. Se l'host non risponde le cause più comuni sono la presenza di un *firewall*[16] o problemi sul cavo di rete o scheda di rete.

- Utilizzo di *netstat* per informazioni statistiche sulla rete, quali lo stato dei socket. Per visualizzare tutte le porte TCP in ascolto o collegate nel computer, da Shell digitare `netstat -nap tcp` ottenendo ad esempio il seguente output, (abbiamo evidenziato in grassetto un Host-k in ascolto).

```
Active Connections
  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:5000           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:8080           0.0.0.0:0              LISTENING
  TCP    127.0.0.1:1027         0.0.0.0:0              LISTENING
  TCP    127.0.0.1:1027         127.0.0.1:2446         TIME_WAIT
  TCP    131.114.9.162:139      0.0.0.0:0              LISTENING
  TCP    131.114.9.162:2374     131.114.9.61:80        CLOSE_WAIT
  TCP    131.114.9.162:2404     195.210.93.140:80      TIME_WAIT
```

L'output del comando mostra le connessioni *attive*, *in attesa* e in *chiusura* e per ognuna di esse riporta il protocollo, l'indirizzo locale con la porta, l'indirizzo remoto con la porta e lo stato della connessione rispetto alla porta locale. Lo stato può essere:

```
LISTENING:       la porta è aperta in ascolto ed attende una connessione.
SYN_SENT:        la porta è aperta e sta inviando dati.
SYN_RECEIVED:    la porta è aperta e sta ricevendo dati.
ESTABLISHED:     la porta è aperta ed è collegata con un computer.
TIME_WAIT:       la connessione è terminata, tuttavia la porta attende la chiusura.
FIN_WAIT_1:      la porta è ancora aperta, ma si accinge a chiudersi.
FIN_WAIT_2:      la porta è ancora aperta, ma si accinge a chiudersi.
CLOSE_WAIT:      la porta attende la chiusura.
CLOSING:         la porta è in chiusura.
LAST_ACK:        la porta è in chiusura (ultimo messaggio prima di chiuderla).
CLOSED:          la porta è effettivamente chiusa.
```

Non tutti questi stati sono facili da vedere poiché la loro permanenza può durare pochi decimi di secondo. Alcuni programmi di protezione, ad esempio un *firewall*, possono forzare alcuni di questi stati, anticipandoli o bloccandoli. In particolare, lo stato `TIME_WAIT` è spesso causa di errori di funzionamento dei programmi client. Quando un programma client, utilizzando una porta x si connette ad un programma server sulla porta y ed in seguito alla fine del lavoro si disconnette (volontariamente o meno) e prova a ricollegarsi continuando ad utilizzare la porta x si verifica un errore. Infatti, subito dopo la chiusura di una connessione, la porta x va nello stato `TIME_WAIT`, risultando occupata per qualche secondo. Il tentativo di riconnettersi mediante la stessa porta x genererà un errore di "Indirizzo in uso". Per risolvere questo problema è necessario attendere la completa chiusura della porta oppure cambiare la porta locale x con un'altra libera. In genere basta specificare la porta locale 0 per farsi assegnare un socket ad una porta libera.

[14] Questi tool, al pari degli strumenti a riga di comando della directory bin di Java, sono piccole applicazioni eseguibili fornite dai sistemi operativi in opportune cartelle. Ad esempio nel caso di *Microsoft Windows XP* © si trovano in `C:\WINDOWS\system32`, mentre per le distribuzioni di *Linux* ® o *FreeBSD* ® si trovano in `/sbin` o `/bin`. Esiste uno standard (IEEE – POSIX, *Portable Operating System Interface*, la *X* sta per uniX) che specifica la sintassi e la collocazione dei comandi di sistema, ma purtroppo non è seguito, per cui vi sono differenze anche tra sistemi basati su Unix.

[15] Il termine *ping* fu originariamente coniato nel gergo dei sommergibilisti per indicare il suono di un impulso sonar che ritornava dopo aver colpito un oggetto. Il *ping* valuta la raggiungibilità degli host attraverso uno o più piccoli pacchetti (datagrammi, a livello IP) di rete.

[16] Il *firewall* (letteralmente "muro tagliafuoco") è un'applicazione che protegge un host da accessi abusivi, bloccando l'apertura di porte sia in ingresso che in uscita; ad esempio si possono filtrare i pacchetti in ingresso che arrivano da un dato indirizzo IP, oppure i pacchetti in uscita provenienti da una data applicazione (client di posta, web browser,...). Firewall scaricabili gratuitamente sono *Zone Alarm* (http://www.zonelabs.com) e *Kerio Personal Firewall* (http://www.kerio.com/kpf_home.html).

ESERCITAZIONE TIGA: componenti Swing, applet

Il settore finanze di un ente pubblico territoriale decide di integrare il proprio portale web con un servizio di "calcolatrice" che, visualizzata in un frame laterale accanto ai moduli per il calcolo delle imposte, permetta ai contribuenti di eseguire semplici operazioni di calcolo.

In particolare, il committente sottolinea la possibilità che in un futuro prossimo vengano aggiunte ulteriori operazioni.

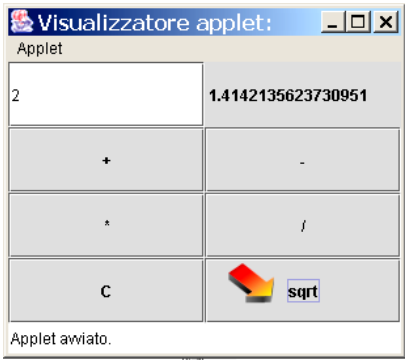


Fig.1 – Applet calcolatrice

1) Sviluppare[17] una applet dotata di interfaccia grafica come in figura, che realizza il servizio di calcolatrice a notazione postfissa.

In alto a destra viene visualizzato – come etichetta – il valore attuale dell' **accumulatore**; in alto a sinistra – in un campo di testo modificabile – il valore dell' **operando.**

Le operazioni binarie disponibili sono la somma (**+**), la sottrazione (-), la moltiplicazione (*) e la divisione (**/**); quelle unarie, la radice quadrata (**sqrt**) e l'azzeramento (**C**). Tutte le operazioni lasciano inalterato l'operando e collocano il risultato nell'accumulatore. Nel caso di operazione unaria $\boldsymbol{U}(\bullet)$, il nuovo valore dell'accumulatore diventa:

```
accumulatore = U(accumulatore).
```

Nel caso di operazione binaria $(\bullet)\boldsymbol{B}(\bullet)$, il nuovo valore dell' accumulatore diventa:

```
accumulatore = (accumulatore)B(operando).
```

2) Eseguire un intervento di *manutenzione perfettiva*, consistente nell'aggiunta di una nuova operazione, ad esempio l'operazione unaria di negazione (**neg**). Quante istruzioni sono state modificate o aggiunte in totale ?

3) Inserire dei valori inconsistenti (stringhe alfanumeriche) come valore dell'operando, ed eseguire una operazione binaria. Quali eccezioni vengono generate? Eseguire delle operazioni matematicamente indeterminate, quali la divisione per zero. Cosa accade? Elencare le principali anomalie. L'operazione di azzeramento si potrebbe matematicamente ottenere sommando all'accumulatore il suo opposto. Esistono degli stati dai quali non è possibile uscire senza l'azzeramento?

SUGGERIMENTI:

Si consiglia di gestire in modo unitario gli eventi originati dai sei tasti, definendo un singolo ascoltatore per gli eventi di azione. All'interno del metodo `actionPerformed` vengono invocati due servizi, relativi all'applicazione. Il primo servizio serve ad impostare un opportuno *stato* a seconda del bottone pigiato (riferibile come argomento attraverso `getSource()`), mentre il secondo servizio esegue una corrispondente operazione della calcolatrice, aggiornando contestualmente gli oggetti grafici dell'interfaccia, a seconda dello *stato* attuale.

Si osservi che l'editazione di un campo di testo *non* necessita di alcuna gestione di eventi da parte del programmatore, in quanto inclusa nel comportamento predefinito dell'oggetto.

La stringa contenuta nel campo di testo può essere prelevata con il metodo `String getText()`.

La stringa contenuta nella etichetta può essere modificata con il metodo `setText(String s)`.

[17] D'ora in poi i adoperi l'ambiente *netBeans IDE*® per l'editazione e la compilazione del codice. Per l'esecuzione delle applet si usi l'applicativo *appletviewer*® oppure un web browser. Nel caso di *applet viewer*®, eventuali eccezioni vengono stampate sulla finestra dei comandi da cui l'applicazione è stata avviata. Nel caso del web browser le eccezioni possono essere visualizzate nella *Java console*. Tale finestra appare al caricamento di una applet se nel *Java*® *plug-in Control Panel* (i) è stato impostato *Basic* → *Java Console* → *Show Console* ed (ii) il pannello *Browser* → *Settings* contiene il proprio web browser opportunamente selezionato.

Progettazione:

Per poter gestire in modo opportuno le interazioni dell'utente con i componenti attivi di una interfaccia, che possono verificarsi in diverse condizioni di funzionamento, è necessario che l'applicazione tenga traccia delle operazioni significative compiute fino a quel momento[18]. Per questo motivo un'applicazione *guidata dagli eventi* (*event-driven*) deve gestire al proprio interno un concetto di *stato*, cioè una serie di indicatori che specifichino in quale contesto l'applicazione si trovi al momento.

Nella fattispecie, la gestione dell'evento di azione sui bottoni può essere schematizzata come in Fig.1. L' *Utente* inserisce un numero (nel campo di testo *JTextField*), quindi attiva un bottone (oggetto *b JButton*) il quale *non sa* cosa avverrà come conseguenza; questa azione viene recepita da *AscoltatorePremiBottone* attraverso l'invocazione di *actionPerformed(evento e)*. Come conseguenza di ciò, *MiaInterfaccia* riceverà due chiamate consecutive: *individuaStato(b)*, che cambia lo stato, ed *eseguiOperazione()*, che in base allo stato esegue un'opportuna elaborazione, provvedendo anche a prelevare l'operando (il contenuto dell'oggetto *JTextField*) ed a restituire il risultato (la modifica dell' oggetto *JLabel*). Infine, l' *Utente* può leggere il risultato osservando lo schermo.

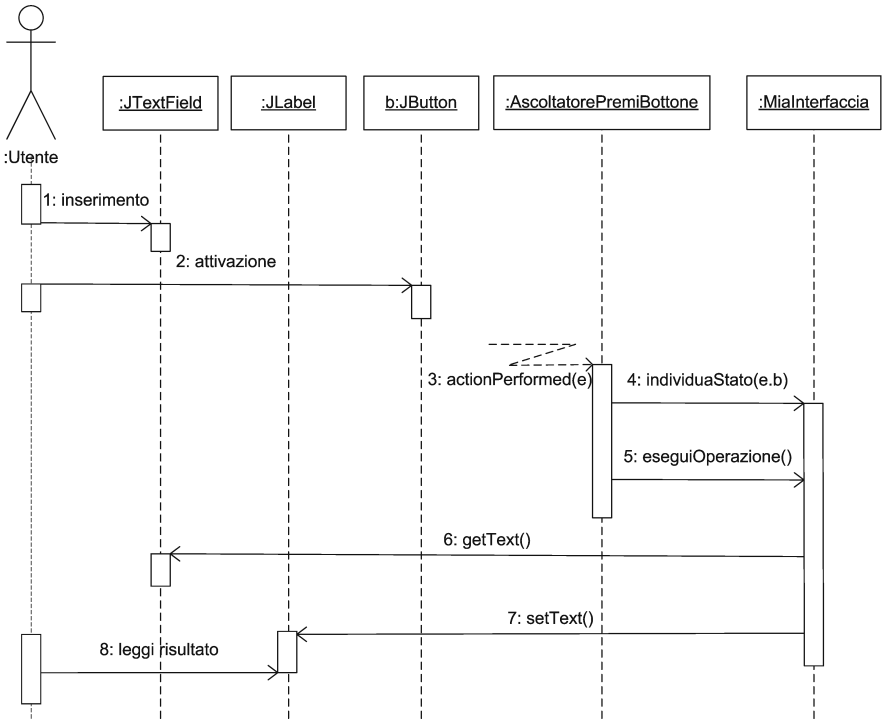


Fig.1 – Diagramma di sequenza per la gestione degli eventi vista dal programmatore.
In UML i due punti prima del nome della classe indicano una istanza di quella classe, ossia un oggetto.

Codifica in linguaggio Java:

La *manutenzione perfettiva* costituisce la parte prevalente (50%) del costo di manutenzione del software, e consiste nel miglioramento delle funzionalità. Le necessità di evoluzione di un'applicazione sono un fatto spesso connaturato con le applicazioni software. In particolare, specie nelle applicazioni che stanno a contatto diretto con utenti finali inesperti, le applicazioni sono chiamate a evolvere, tanto più quanto più esse risultano utilizzate in modo proficuo. Infatti, un'applicazione usata nel lavoro quotidiano delle persone, nata dall'esigenza di automatizzare o facilitare alcune procedure di lavoro, non appena effettivamente usata, provoca dei riflessi sul modo di lavorare che possono indurre modifiche di tipo organizzativo e che, a loro volta, portano a richiedere modifiche all'applicazione. Nasce così una classica situazione di retroazione: l'organizzazione del lavoro fa nascere esigenze di automazione mediante applicazioni software le quali, provocando una modifica nell'organizzazione del lavoro, inducono ulteriori modifiche. E così via.

La tipica struttura di un'applicazione guidata dagli eventi prevede una variazione nello stato, come conseguenza di un evento, e l'invocazione di una struttura di controllo composta da una o più istruzioni `switch`, che attivano la parte di codice relativa allo stato corrente. Quando è necessario distinguere diversi stati per un certo evento, strutturando gerarchicamente lo stato è possibile che ciascun `case` contenga un metodo che identifica a sua volta uno stato di "secondo livello" in base al quale determinare il codice per il servizio da eseguire.

[18] Per esempio, in un editor grafico per disegni geometrici la pressione del pulsante del mouse può indicare diverse azioni a seconda che in corrispondenza del puntatore vi sia un oggetto (quindi *selezione dell'oggetto*) o meno (*deselezione degli oggetti selezionati*), o si sia precedentemente selezionato uno strumento per tracciare curve (*tracciamento di un punto sulla tela*).

Nella soluzione proposta, l'aggiunta di una nuova operazione implica esclusivamente l'aggiunta di un `case` (con l'implementazione del servizio) e di un elemento nell'array delle etichette dei bottoni. Il gestore di layout provvede automaticamente a collocare il nuovo bottone "allungando" la calcolatrice.

Si osservi il trucco adoperato nel metodo *eseguiOperazione()*: aggiungendo una quantità infinitesima all'operando, si evita di dividere per un operando eventualmente nullo (`accumulatore /= operando+Double.MIN_VALUE`)

```
<!--pagina.html-->
<APPLET CODE="MiaInterfaccia.class" WIDTH=300 HEIGHT=200></APPLET>

// AscoltatorePremiBottone.java
import java.awt.event.*;
import javax.swing.*;

public class AscoltatorePremiBottone implements ActionListener {

    public AscoltatorePremiBottone(MiaInterfaccia interfaccia) {
        interf = interfaccia;
    }

    public void actionPerformed(ActionEvent e) {
        interf.individuaStato((JButton)e.getSource());
        interf.eseguiOperazione();
    }

    private MiaInterfaccia interf;
}

// MiaInterfaccia.java

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

public class MiaInterfaccia extends JApplet {

    public void init() {
        accumulatore = 0.0;
        stato_op = -1;
        inizializzaGUI();
    }

    private void inizializzaGUI() {
        JPanel pannello = new JPanel();

        pannello.setLayout( new GridLayout(1+(OPERAZIONI.length+1)/2,2) );

        operando_view = new JTextField("0.0");
        accumulatore_view = new JLabel(" 0.0");
        operazioni_view = new JButton[OPERAZIONI.length];
        for (int i=0; i<operazioni_view.length; i++)
            operazioni_view[i] = new JButton(OPERAZIONI[i]);

        // crea i gestori di eventi
        ActionListener gestore = new AscoltatorePremiBottone(this);

        // associa i componenti ai rispettivi gestori
        for (int i=0; i<operazioni_view.length; i++)
            ( operazioni_view[i] ).addActionListener(gestore);

        // attacca i compomenti al pannello
        pannello.add(operando_view);
        pannello.add(accumulatore_view);
        for (int i=0; i<operazioni_view.length; i++)
            pannello.add(operazioni_view[i]);

        // attacca il pannello alla Frame
        getContentPane().add(pannello);

    }

    public void eseguiOperazione() {
        double operando =  Double.parseDouble(operando_view.getText());
        switch(stato_op) {
            case 0:    accumulatore += operando;                 break;
            case 1:    accumulatore -= operando;                 break;
            case 2:    accumulatore *= operando;                 break;
            case 3:    accumulatore /= operando+Double.MIN_VALUE; break;
            case 4:    accumulatore = 0.0;                       break;
            case 5:    accumulatore = Math.sqrt(accumulatore);   break;
            //case 6:    accumulatore = -accumulatore;             break;
```

```
            default:
        }
        accumulatore_view.setText(" " + Double.toString(accumulatore));
    }

    public void individuaStato(JButton b) {
        for (int i=0; i<operazioni_view.length; i++)
            if ( b == operazioni_view[i] ) {
                stato_op = i;
                break;
            }
        return;
    }

    // campi interfaccia
    private JButton[]               operazioni_view;
    private JLabel                  accumulatore_view;
    private JTextField              operando_view;

    private int                     stato_op;
    private double                  accumulatore;
    private static final String[]   OPERAZIONI = {
        "+", "-", "*", "/", "C", "sqrt", //"neg"
    };
}
```

Testing del programma:
Osservare le eventuali eccezioni durante il test

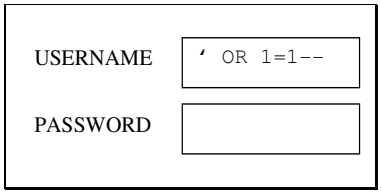
OPERANDO	ACCUMULATORE	OPERAZIONE	NUOVO ACCUMULATORE
0.0	0.0	qualsiasi	0.0
qualsiasi	2.0	sqrt	1.4142135623730951
qualsiasi	qualsiasi	C	0.0
1.0	0.0	+	1.0
0.0	1.0	/	Infinity
0.0	Infinity	qualsiasi, tranne * e C	Infinity
0.0	Infinity	*	NaN
qualsiasi	NaN	qualsiasi, tranne C	NaN
qualsiasi	NaN	C	0.0
qualsiasi	-2.0	sqrt	NaN
12ab	qualsiasi	qualsiasi	inalterato (java.lang.NumberFormatException: For input string: "12ab")
	qualsiasi	qualsiasi	inalterato (java.lang.NumberFormatException: empty string)

Nella progettazione di interfacce grafiche, occorre avere dei meccanismi per impedire che l'utente (accidentalmente o intenzionalmente) inserisca dati in ingresso anomali, o comunque non previsti dal programmatore, le cui conseguenze sono imprevedibili. Vediamo un esempio di 'iniezione' di codice da parte di un hacker.

Si supponga che un' applicazione possa eseguire, a tempo di esecuzione, dei comandi SQL (tecnologia nota come "dynamic SQL") e che si abbia un' interfaccia di login, la quale chiede due stringhe: username e password. Il programmatore ha previsto di costruire una stringa in linguaggio SQL, per poi interrogare la base di dati e controllare se l'utente è registrato:

```
String istruzione =       "SELECT COUNT(*) FROM UTENTI WHERE USERNAME = '" +
                          username + "' AND PASSWORD = '" + password + "'";
```

L'hacker inietta i seguenti frammenti di codice SQL:



la stringa risultante diventa:

```
SELECT COUNT(*) FROM UTENTI WHERE USERNAME = '' OR 1=1 --' AND PASSWORD = ''
```

Il frammento `OR 1=1` provoca che `COUNT(*)` ritorni sempre un numero positivo (se nella tabella c'è almeno una riga), ed il doppio tratto `--` indica un commento SQL, per cui il resto dello statement (visualizzato in italico) verrà ignorato!

Inoltre, se il sistema di connessione al database permette l'esecuzione di più statement, in SQL separati da "**;**", e supponendo che inavvertitamente l'applicazione abbia i diritti per cancellare tabelle o chiudere il database, è possibile inserire i seguenti username, dagli effetti catastrofici:

```
' OR 1=1;UPDATE PREZZI SET COSTO = 0--
' OR 1=1;SHUTDOWN--
' OR 1=1;UPDATE PREZZI SET COSTO = 0;DROP TABLE REVISIONE_CONTI;SHUTDOWN--
```

Un host (*BufferHost*) offre il servizio di *Buffer di stringhe,* per host produttori (*ProducerHost*) e consumatori (*ConsumerHost*), interagenti secondo un modello a scambio di messaggi, mediante l'uso del protocollo di trasporto TCP (Fig.1).

Ciascun host produttore invia un numero P (parametro utente) di stringhe, ad intervalli random di 3÷5 secondi l'una dall'altra, sulla porta 8080 del *BufferHost*. Ciascun host consumatore richiede un numero C (parametro utente) di stringhe, ad intervalli random di 3÷5 secondi tra una precedente ricezione e la successiva richiesta, sulla porta 8080 del *BufferHost*.

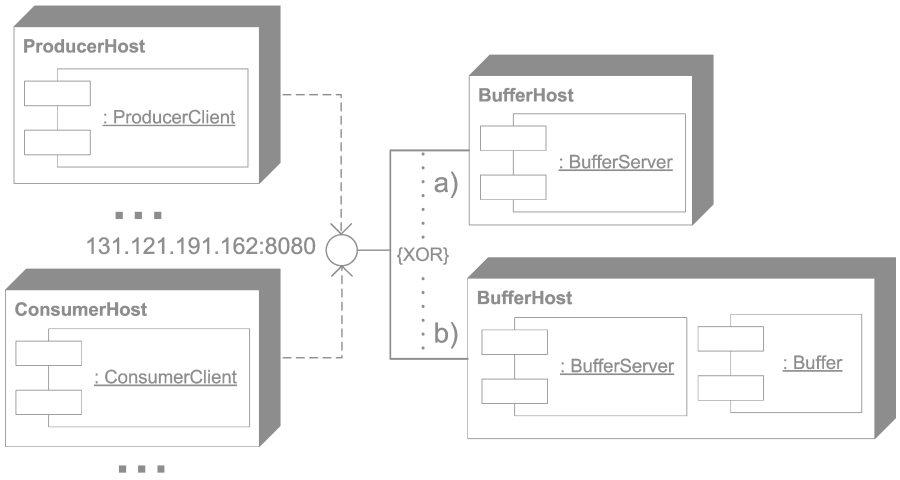


Fig.1 – Distribuzione dei componenti nel caso di Server senza thread (a) e con thread (b)

Sviluppare in Java le classi `ProducerClient`, `ConsumerClient` e `BufferServer` che, distribuite sui nodi della rete come raffigurato in Fig.1, realizzino quanto suddetto. Per le attese del consumatore e produttore, usare il metodo (java.lang.Thread)

```
public static void sleep(long millis) throws InterruptedException
```

Durante l'attesa di un `ConsumerClient` dovuta a buffer vuoto, si possono implementare due diverse gestioni delle risorse di rete, a seconda dei tempi medi di attesa e della disponibilità di risorse del sistema. Nella prima gestione, il canale di comunicazione viene chiuso, ed il `ConsumerClient` si pone in ascolto su una porta concordata, in attesa della stringa. Nella seconda modalità, di più semplice realizzazione, il canale di comunicazione viene mantenuto in attesa della risposta del `BufferServer`.

Si realizzi `BufferServer` in due diverse modalità: (a) senza l'uso di thread e (b) con i thread, secondo le seguenti indicazioni.

a) TCP Server

Si supponga per semplicità che il buffer sia *illimitato*, e che le richieste sopraggiunte a coda vuota siano poste in una coda FIFO[19] e servite appena possibile. Per gestire richieste ravvicinate sulla porta di ascolto del `BufferServer` si usi seguente costruttore (java.net.ServerSocket), dove l'intero `backlog`[20] indica il numero massimo di richieste TCP che possono essere automaticamente accettate ed accodate senza che la `accept()` sia ancora stata invocata.

```
public ServerSocket(int port, int backlog) throws IOException
```

Se la coda di backlog diventa piena, un tentativo di connessione sulla corrispondente porta di ascolto del BufferHost viene rifiutato. Catturare l'eccezione generata e gestirla provando a riconnettersi ad intervalli random di 5÷7 secondi; al terzo tentativo consecutivo fallito, terminare l'esecuzione del produttore/consumatore relativo.

b) TCP Server multi-thread

Si supponga che il buffer sia *limitato*, e si riprogetti il sottosistema *BufferHost* adoperando i thread per assegnare, ad ogni richiesta di servizio, un flusso sequenziale di controllo indipendente, provvedendo a gestire in mutua esclusione gli accessi multipli al buffer e sincronizzando tra loro gli accessi di produttori e consumatori.

In particolare, la classe `Buffer` implementa il buffer FIFO di dimensione limitata, con i metodi `put/get` che garantiscono ai thread un accesso mutuamente esclusivo e sincronizzato, evitando il blocco reciproco di tutte le attività del sistema (deadlock) o l'attesa indefinita di una di esse (starvation). Nella classe `BufferServer`, il metodo main esegue un loop infinito in cui accetta richieste di connessione su una porta e – per ognuna di esse – crea un thread per gestirle. In tal caso non è necessaria la gestione della coda di backlog, in quanto richieste ravvicinate sulla porta di ascolto sono immediatamente gestite mediante thread autonomi, pertanto il flusso di controllo relativo a `BufferServer` é quasi sempre bloccato sulla `accept`.

Mentre i consumatori rimangono in attesa del messaggio prima di inviare la prossima richiesta, i produttori inviano incessantemente messaggi, causando un aumento dei thread sospesi, nel caso di buffer pieno. Visualizzare, ad ogni richiesta di connessione accettata, il numero di thread sospesi, includendo in un `ThreadGroup` ogni thread creato, quindi adoperando il metodo `activeCount()` per conoscere il numero di thread attivi del gruppo. Sebbene i thread abbiano un costo inferiore ai processi, un loro aumento indefinito può esaurire le risorse di sistema. Quali grandezze dell'applicazione influenzano tale aumento ?

[19] *First In, First Out*, ossia il primo elemento ad entrare sarà il primo ad uscire.

[20] *Backlog* è, letteralmente, il cumulo di lavoro arretrato. È possibile che, dopo aver eseguito la accept(), il server debba eseguire altre operazioni e che, durante queste operazioni, sopraggiungano altre richieste prima che il server possa rieseguire la accept().

ESERCITAZIONE TIGA: TCP Client, TCP Server … – SOLUZIONE PROPOSTA

Progettazione[21]:

I parametri da passare a `ProducerClient`/`ConsumerClient` sono il numero di stringhe da produrre/consumare e l'indirizzo IP del `BufferServer`. Nel primo caso viene anche fornita la stringa base con cui formare le stringhe numerate.

Per permettere al `BufferServer` il riconoscimento del tipo di host connesso sulla porta 8080 (`Producer` o `Consumer`) vi sono le classi `StartProd` e `StartCons` che, senza alcun metodo o campo, servono semplicemente per inviare nel canale un oggetto iniziale il cui tipo è riconoscibile mediante `instanceof`. Si osservi che per deserializzare un oggetto non occorre conoscere la classe, ma basta eseguire un cast generico ad `Object`.

Il produttore si limita a creare un socket, associarvi un flusso di uscita sul quale inviare il pacchetto `StartProd`, quindi scrivere nel flusso l'oggetto prodotto. Il consumatore crea i flussi in entrambe le direzioni, in quanto occorre inviare il pacchetto `StartCons`, quindi ricevere l'oggetto con una `readObject()` bloccante (timeout = 1 ora). La gestione delle eccezioni è ridotta al minimo, tranne che per la `ConnectException` che segnala il rifiuto del server ad aprire una connessione (es. a seguito di server spento o di backlog piena) e – alla terza volta consecutiva – produce la terminazione dell'applicazione. Questo effetto viene realizzato mediante un contatore (`numExc`) di eccezioni, azzerato ad ogni esecuzione completa del blocco `try`.

Infatti, in caso di eccezione in un punto del blocco `try` le rimanenti istruzioni del blocco non verranno mai più eseguite. Dopo aver generato un oggetto eccezione, il sistema cerca nei blocchi `catch` se esite una classe alla quale tale oggetto appartiene. Se esiste, viene eseguito solo il codice di tale blocco `catch`, e poi l'esecuzione riprende dal blocco `finally` o, se non presente, dalle istruzioni dopo la sequenza `try-catch`. Se invece nessun blocco `catch` prevede una classe/superclasse [22] per l'oggetto eccezione generato, allora viene comunque eseguito il blocco `finally` [23], ma poi l'esecuzione del metodo abortisce. Quindi vengono liberate le risorse relative alla esecuzione del metodo, senza ritornare al metodo chiamante alcun valore eventualmente previsto, ma producendo una eccezione nell'ambiente di esecuzione del metodo chiamante, il quale può aver gestito tale invocazione in un blocco `try-catch` o meno. Se, nella catena di invocazioni, nessun metodo chiamante ha previsto un blocco `catch` per catturare l'eccezione, allora anche il metodo `main`, da ultimo, abortisce e la JVM termina la propria esecuzione.

Nell'implementazione del server, occorre considerare separatamente due alternative.

a) *Senza l'uso dei thread.* La classe `BufferHost` implementa il Buffer e la coda di flussi in attesa, mediante la classe `LinkedList` che può essere gestita come una coda infinita di oggetti qualsiasi. Infatti, se la coda è vuota, il server mantiene la connessione con il consumatore conservandone il relativo flusso di uscita, sul quale verrà successivamente inoltrata la stringa. Si osservi che, quando vi sono dei consumatori in attesa (cioè il buffer è vuoto), il dato prodotto viene immediatamente inoltrato al primo di essi, e non inserito nel buffer. Per riempire la coda di backlog basta togliere il commento alla riga contrassegnata da (*) in `BufferServer`, così impegnando il server in un lungo ciclo for [24].

b) *Con multithreading*. La classe `BufferServer` adopera i gruppi di thread per manipolare un insieme di thread tutti in una volta, come se fossero un unico thread, ad esempio per cambiarne la priorità.

```
ThreadGroup(String name)                    // crea un nuovo gruppo di thread
Thread(ThreadGroup group, String name)      // crea un nuovo thread in un gruppo
public int activeCount()                    // numero di thread attivi nel gruppo
```

Per risvegliare i thread viene adoperata la `notifyAll` (che risveglia tutti i thread), più costosa della `notify` (che ne risveglia uno a caso) ma più sicura. Supponiamo di sostituire nel codice la `notifyAll` con la `notify`[25]. Sappiamo che i thread bloccati possono essere di due tipi (produttore e consumatore) e che, in alcuni stati particolari del buffer (pieno/vuoto), risvegliarne uno a caso potrebbe significare risvegliare il tipo sbagliato. Un produttore a buffer pieno, o un consumatore a buffer vuoto, appena risvegliati si ri-bloccano immediatamente senza risvegliare un altro thread con una `notify`. Ciò significa che, se viene risvegliato il thread sbagliato, solo una nuova richiesta esterna può sbloccare la situazione. Si supponga che un produttore p_0, bloccato perché al suo arrivo iniziale il buffer era pieno, non venga risvegliato dalla `notify` per un tempo indefinito[26]. Dopo un certo numero di consumatori/produttori in arrivo o risvegliati, il buffer diviene vuoto ma in tutte le `notify` non viene mai selezionato il produttore p_0. Pur essendo l'unico produttore sospeso, l'arrivo di un nuovo consumatore non può provocarne il risveglio, dal momento che questi viene immediatamente sospeso poiché la coda è vuota. Solo un nuovo produttore *potrebbe* risvegliarlo, oppure provocarne indirettamente il risveglio dopo aver risvegliato un consumatore. In conclusione, p_0 potrebbe bloccarsi per un tempo indefinito (starvation).

---

[21] Per i metodi relativi alle connessioni TCP ed il passaggio di oggetti tra processi, vedere l'esercitazione sui Socket.

[22] Se vi sono due o più blocchi catch compatibili con l'oggetto eccezione, allora verrà eseguito solo il primo (in ordine di scrittura) di essi. Quindi le sottoclassi vanno dichiarate prima delle rispettive superclassi, altrimenti non verrebbero mai selezionate.

[23] Il blocco *finally* viene eseguito comunque (se viene eseguita la clausola *try*) anche se non vi sono eccezioni (e quindi viene eseguito l'intero codice del blocco *try*). Serve ad evitare duplicazione di codice e garantire alcune operazioni che non verrebbero eseguite in caso di eccezione non catturata (in cui l'intero metodo abortisce). Tipicamente è usata per operazioni di pulizia finale durante I/O.

[24] In tal caso non si può adoperare una *sleep* per ritardare l'esecuzione delle *accept()*, in quanto si produrrebbe un passaggio del processo server in stato "bloccato" e quindi una interruzione del funzionamento delle relative risorse, tra cui l'oggetto ServerSocket e la relativa coda di backlog.

[25] Si osservi che, sia nella istruzione *notify* che *notifyAll*, i nuovi thread potranno accedere al buffer solo dopo che il thread che le ha invocate è uscito dal blocco di codice sincronizzato che comprende l'istruzione *notify* o *notifyAll*. Per lo stesso motivo, nella *notifyAll*, tutti i thread risvegliati potranno accedere uno alla volta.

[26] La casualità non garantisce che un particolare thread venga risvegliato dopo un qualsiasi numero N di tentativi. Mentre il fatto di risvegliare tutti i thread e rimetterli in competizione, garantisce che – seppur nella competizione possa essere il thread sbagliato a gestire per primo il buffer – subito dopo il thread giusto potrà operare. Infatti tutti i thread risvegliati avranno la possibilià di accedere alla risorsa, sebbene in un ordine non prevedibile perché aventi la medesima priorità.

Questo esempio mostra chiaramente che nell'uso dei thread spesso conviene procedere per *pattern*, ossia ricondurre la gestione di un problema specifico a modelli generali ben noti (produttore-consumatore è uno di questi modelli), piuttosto che reimplementare soluzioni che richiederebbero un testing oneroso per essere verificate[27], anche se i costrutti offerti da Java sono un compromesso tra efficienza e facilità di programmazione.

Si osservi che la gestione dei thread in arrivo è FIFO fino a che viene sospeso al massimo un solo thread; altrimenti il non determinismo implicito nel risveglio multiplo della `notifyAll` rende l'ordine di servizio indipendente dall'ordine di arrivo. Pertanto è possibile che le stringhe prodotte con un numero progressivo da un produttore, vengano memorizzate con altro ordine (Fig.2).

Un thread termina "naturalmente", quando raggiunge la fine del metodo `run()`. Metodi a terminazione forzata, come `stop()`, `suspend()`, `resume()`, `destroy()`, sono **`deprecated`** in quanto non danno modo al thread di rilasciare risorse condivise, quindi sono causa di potenziali deadlock, oppure consentono il rilascio delle risorse in stato inconsistente (non previsto dal programmatore).

Codifica in linguaggio Java:

```
// StartProd.java
import java.io.*;
public class StartProd implements Serializable {}

// StartCons.java
import java.io.*;
public class StartCons implements Serializable {}

// ProducerClient.java
import java.io.*;
import java.net.*;
import java.util.*;

public class ProducerClient {
    public static void main(String args[]) {
        final String        DATO            = args[0];
        final int           NUM_DATI        = Integer.parseInt(args[1]);
        final String        INDIRIZZO_IP    = args[2];
        final int           PORTA           = 8080;
        int                 numExc          = 0;

        InetAddress         ind             = null;
        Socket              socket          = null;
        ObjectOutputStream  oout            = null;
        ObjectInputStream   oin             = null;

        for (int i=0; i<NUM_DATI; i++) {
            try {
                ind = InetAddress.getByName(INDIRIZZO_IP);
                socket = new Socket(ind, PORTA);
                oout = new ObjectOutputStream (socket.getOutputStream());
                oout.writeObject(new StartProd());

                oout.writeObject(DATO + i);
                System.out.println("inviato: " + DATO + i);
                long base = (numExc > 0) ? 5000 : 3000;
                java.lang.Thread.sleep( base + (long)(Math.random()*2000.0));
                numExc = 0;
            }
            catch (IOException e) {
                if (e instanceof ConnectException) {
                    numExc++;
                    System.err.println("Connessione rifiutata (" + numExc + ")");
                    if (numExc >= 3) {
                        System.err.println("Terminazione per problemi di connessione.");
                        System.exit(1);
                    }
                }
                else
                    e.printStackTrace();
            }
            catch (InterruptedException e) { e.printStackTrace(); }
            finally {
                try {
                    if (numExc==0) {
                        oout.close();
                        socket.close();
                    }
                }
                catch (IOException e) { e.printStackTrace(); }
            }
        }
    }
}
```

[27] Oltre ai thread, un altro meccanismo che consente di scrivere del codice elegante ma di difficile verifica è quello della ricorsione.

```
// ConsumerClient.java
import java.io.*;
import java.net.*;
import java.util.*;

public class ConsumerClient {
    public static void main(String args[]) {
        final int           NUM_DATI        = Integer.parseInt(args[0]);
        final String        INDIRIZZO_IP    = args[1];
        final int           PORTA           = 8080;
        final int           TOUT_SEC        = 3600;

        int                 numExc          = 0;
        InetAddress         ind             = null;
        Socket              socket          = null;
        ObjectOutputStream  oout            = null;
        ObjectInputStream   oin             = null;

        for (int i=0; i<NUM_DATI; i++) {
            try {
                ind = InetAddress.getByName(INDIRIZZO_IP);
                socket = new Socket(ind, PORTA);
                socket.setSoTimeout(TOUT_SEC*1000);
                oout = new ObjectOutputStream (socket.getOutputStream());
                oin = new ObjectInputStream (socket.getInputStream());
                oout.writeObject(new StartCons());

                System.out.println("prelevato: " + (String)oin.readObject());
                long base = (numExc > 0) ? 5000 : 3000;
                java.lang.Thread.sleep(base + (long)(Math.random()*2000.0));
                numExc = 0;
            }
            catch (IOException e) {
                if (e instanceof ConnectException) {
                    numExc++;
                    System.err.println("Connessione rifiutata (" + numExc + ")");
                    if (numExc >= 3) {
                        System.err.println("Terminazione per problemi di connessione.");
                        System.exit(1);
                    }
                }
                else
                     e.printStackTrace();
            }
            catch (InterruptedException e)   { e.printStackTrace(); }
            catch (ClassNotFoundException e) { e.printStackTrace(); }
            finally {
                try {
                    if (numExc==0) {
                        oout.close();
                        oin.close();
                        socket.close();
                    }
                }
                catch (IOException e) {  e.printStackTrace(); }
            }
        }
    }
}
```

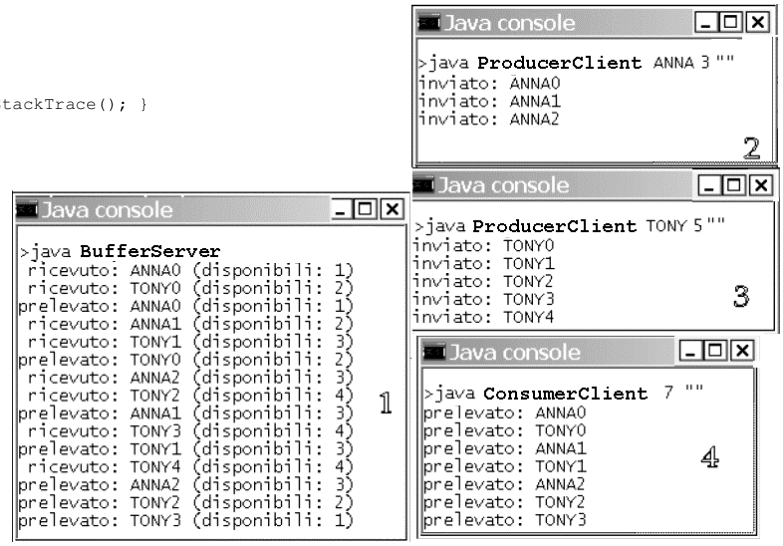


Fig.1 – Scenario di esecuzione standard.

a) TCP Server

```
// BufferServer.java

import java.util.*;
import java.net.*;
import java.io.*;

public class BufferServer {

    public BufferServer() {
        buf = new LinkedList();
        scons = new LinkedList();
    }

    public void add(String s) {
        buf.addLast(s);
    }

    public String remove() {
        return buf.isEmpty() ? null: (String)buf.removeFirst();
    }

    public void addCons(ObjectOutputStream sc) {
         scons.addLast(sc);
    }

    public ObjectOutputStream removeCons() {
        return scons.isEmpty() ? null: (ObjectOutputStream)scons.removeFirst();
    }

    public int size() {
```

```
        return buf.size();
    }

    public int sizeCons() {
        return scons.size();
    }

    public static void main(String[] args) {

        final int           PORTA           = 8080;
        final int           BACKLOG         = 7;
        String              unit            = null;
        ServerSocket        servSock        = null;
        Socket              sock            = null;
        ObjectInputStream   oin             = null;
        ObjectOutputStream  oout            = null,     oout2       = null;
        BufferServer        mioBuf      = new BufferServer();

        try {
            servSock = new ServerSocket(PORTA, BACKLOG);
            while (true) {
                sock = servSock.accept();
                oin  = new ObjectInputStream( sock.getInputStream());
                oout = new ObjectOutputStream( sock.getOutputStream());

                if ( (Object)oin.readObject() instanceof StartCons) { // consumatore
                    unit = mioBuf.remove();
                    if (unit!=null) {
                        System.out.println("prelevato: " + unit +
                                           " (disponibili: " + mioBuf.size() + ")");
                        oout.writeObject(unit);
                        oout.close();
                        oin.close();
                    }
                    else {
                        mioBuf.addCons(oout);
                        System.out.println("\t\t   (in attesa: " +
                                           mioBuf.sizeCons() + ")");
                    }
                }
                else {                                              // produttore
                    unit = (String)oin.readObject();
                    System.out.print(" ricevuto: " + unit );
                    oout2 = mioBuf.removeCons();
                    if (oout2!= null) {
                        oout2.writeObject(unit);
                        oout2.close();
                        System.out.println(" (in attesa: " + mioBuf.sizeCons() + ")");
                    }
                    else {
                        mioBuf.add(unit);
                        System.out.println(" (disponibili: " + mioBuf.size() + ")");
                    }
                    oout.close();
                    oin.close();
                }
                  //for (long i=0; i<Long.MAX_VALUE; i++) ; // (*) riempie coda di backlog
            }
        }
        catch (IOException e) { e.printStackTrace(); }
        catch (ClassNotFoundException e) {  e.printStackTrace(); }
        finally {
            try {
                oin.close();
                oout.close();
                sock.close();
                servSock.close();
            }
            catch (IOException e) {  e.printStackTrace(); }
        }
    }

    private LinkedList buf, scons;
}
```

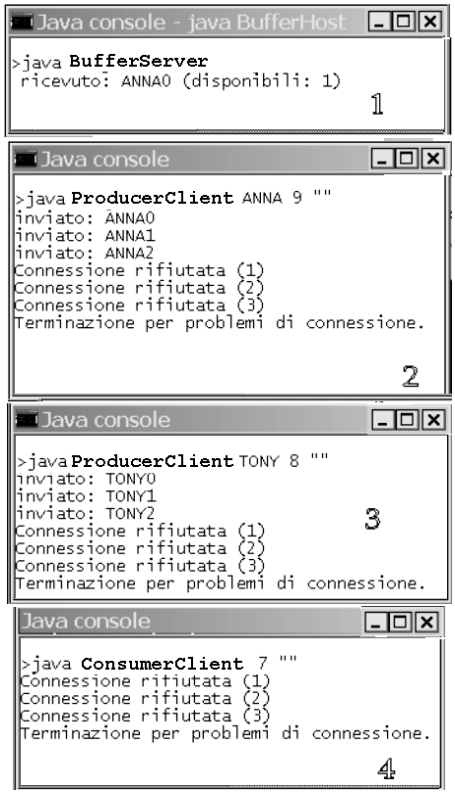


Fig.2 – Scenario di esecuzione con coda di backlog piena.

b) TCP Server multi-thread

```
// Buffer.java

import java.util.*;

public class Buffer {

    private LinkedList  buf;
    private int         maxNumMsg;

    public Buffer(int size) {
        maxNumMsg = size;
        buf = new LinkedList();
    }

    public synchronized void put(String s) {
```

```
        while ( buf.size() >= maxNumMsg )
            try {
                wait();
            }
            catch (InterruptedException e) {}
        buf.addLast(s);
        notifyAll();
    }

    public synchronized String get() {
        String result;
        while ( buf.isEmpty() )
            try {
                wait();
            }
            catch (InterruptedException e) {}
        result = (String)buf.removeFirst();
        notifyAll();
        return result;
    }

    public synchronized int size() {
        return buf.size();
    }
}

// BufferServer.java

import java.net.*;
import java.io.*;

public class BufferServer extends Thread {
    static  final int          PORTA  = 8080;
    private Buffer             buff   = null;
    private Socket             sock   = null;
    private ThreadGroup        group  = null;
    private ObjectInputStream  oin    = null;
    private ObjectOutputStream oout   = null;
    private String             unit   = null;

    public BufferServer(Socket s, Buffer b, ThreadGroup sreq) throws IOException {
        super(sreq,"");
        group = sreq;
        buff = b;
        sock = s;
        oin  = new ObjectInputStream( sock.getInputStream());
        oout = new ObjectOutputStream( sock.getOutputStream());
        start();
    }

    public void run() {
        System.out.println("\t\t\t(rich.attive: " + group.activeCount() + ")");
        try {
            if ( (Object)oin.readObject() instanceof StartCons) { // consumatore
                unit = buff.get();                                  // bloccante
                oout.writeObject(unit);
                System.out.println("prelevato: " + unit +
                                   "\t(disponibili: " + buff.size() + ")");
            }
            else {                                                  // produttore
                unit = (String)oin.readObject();
                buff.put(unit);                                     // bloccante;
                System.out.println(" ricevuto: " + unit +
                                   "\t(disponibili: " + buff.size() + ")");
            }
            oout.close();
            oin.close();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        finally {
            try {
                oin.close();
                oout.close();
                sock.close();
            }
            catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main (String[] args) {
        final int   SIZE = Integer.parseInt(args[0]);

        Buffer          buff     = new Buffer(SIZE);
        ThreadGroup     servReq  = new ThreadGroup("");
        ServerSocket    servSock = null;

        try {
            servSock = new ServerSocket(PORTA);
```

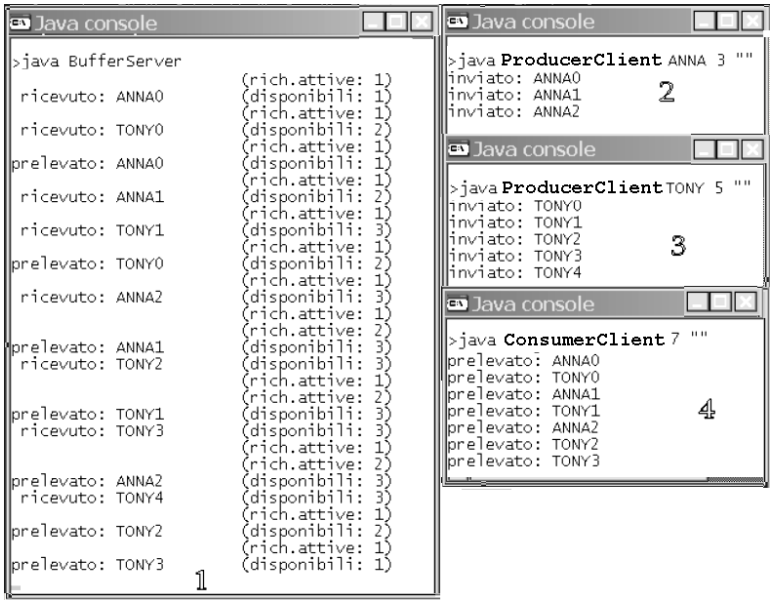


Fig.3 – Scenario di esecuzione standard.

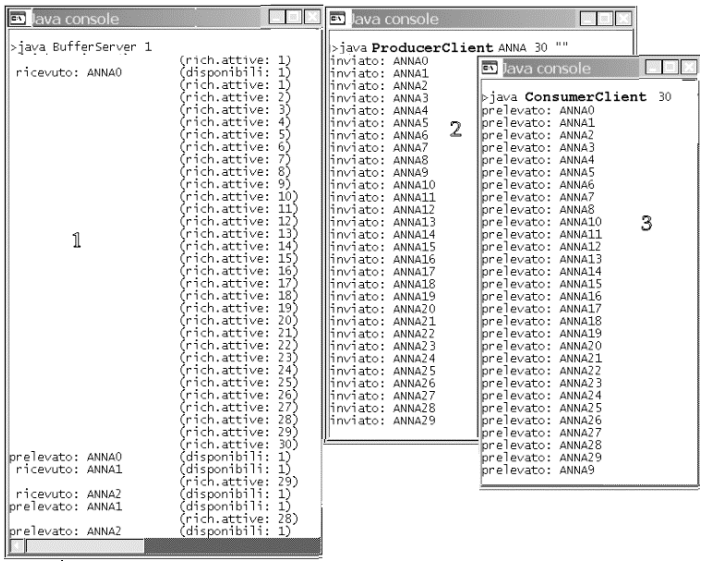


Fig.4 – Scenario di esecuzione con buffer pieno, di dimensione 1. Solo dopo la terminazione del produttore (2), con 29 thread sospesi, è partito il consumatore (3). Si noti che la stringa "ANNA9" è stata l'ultima consumata, pur essendo la nona prodotta.

```
            while (true) {
                Socket sock = servSock.accept();
                try {
                    new BufferServer(sock, buff, servReq);
                }
                catch (IOException e) {
                    sock.close();
                }
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        finally {
            try {
                servSock.close();
            }
            catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Testing del programma:

• Il massimo numero di thread sospesi, dipende dal numero di Produttori e di Consumatori, e dal numero di stringhe che ciascuno di essi produce o consuma. Il numero medio di stringhe nel buffer dipende dalla differenza tra velocità di produzione e di consumo.

• Eseguire due attacchi di tipo DoS (*Denial of Service*) per saturare le risorse del server e rendere il sistema instabile.
a) Nel server senza thread, aumentare la frequenza di produzione/consumazione al limite delle possibilità del sistema (es. 5 msec) in modo da generare nel server una `java.net.SocketException` (indica che c'è stato un errore nel protocollo sottostante, TCP nella fattispecie).
b) Nel server con thread, aumentare la frequenza di produzione/consumazione (non al limite) ed il numero di client (tutti di un solo tipo), in modo da generare l'errore `java.lang.OutOfMemoryError: unable to create new native thread` dopo la generazione di qualche migliaio di thread.

ESERCITAZIONE TIGA: Thread, TimerTask, applicazione del pattern produttore-consumatore al controllo degli approvvigionamenti (compito di esame del 25/06/04)

Un insieme di merci è costituito da diversi tipi di merci e per ogni merce da un certo numero di articoli. La classe Java StockSet realizza un insieme ordinato di merci che ha i seguenti costruttori e metodi

- **public** StockSet(**int** numberOfStocks) che definisce un insieme costituito da un numero di merci pari a numberOfStocks. Il costruttore imposta a zero il numero di articoli di ciascun tipo di merce.
- **public** String toString() che converte uno StockSet in uno String.
- **public boolean** geq(StockSet other) **throws** ArrayIndexOutOfBoundsException che ritorna **true** se questo insieme di merci è maggiore o uguale dell'insieme di merci other specificato come argomento. Un insieme di merci è maggiore o uguale di un altro se, per ogni tipo di merce, il numero di articoli nel primo insieme è maggiore o uguale del numero di articoli nel secondo insieme. I due insiemi devono essere costituiti dallo stesso numero di merci; altrimenti viene sollevata l'eccezione ArrayIndexOutOfBoundsException.
- **public void** order(StockSet ordered) **throws** ArrayIndexOutOfBoundsException che sottrae l'insieme di merci ordered da questo insieme di merci. La sottrazione di un insieme di merci da un altro consiste nel sottrarre, per ciascuna merce, il numero degli articoli del primo da quello del secondo. I due insiemi devono essere costituiti dallo stesso numero di merci; altrimenti viene sollevata l'eccezione ArrayIndexOutOfBoundsException.
- **public void** supply(StockSet supplied) **throws** ArrayIndexOutOfBoundsException che somma l'insieme di merci supplied a questo insieme di merci. La somma di un insieme di merci ad un altro consiste nel sommare, per ciascuna merce, il numero degli articoli del primo a quello del secondo. I due insiemi devono essere costituiti dallo stesso numero di merci; altrimenti viene sollevata l'eccezione ArrayIndexOutOfBoundsException.
- **public void** initRandom(int maxItems) che imposta il numero di articoli di ciascuna merce di questo insieme ad un valore casuale compreso tra 0 e maxItems.
- **public void** numberOfStocks() che ritorna il numero di tipi di merci di questo insieme.

Un *magazzino* è uno StockSet in cui le operazioni di order e supply possono essere eseguite in modo concorrente in accordo alla seguente condizione di sincronizzazione: l'esecuzione di un ordine può essere completata se la merce in magazzino è maggiore di quella ordinata; altrimenti, l'ordine deve essere sospeso fino a quando il magazzino non sarà approvvigionato.

Esercizio n. I. Il candidato realizzi la classe Stockhouse che modella un magazzino

Esercizio n. II. Il candidato realizzi inoltre un timer task di nome Report che va in esecuzione periodicamente ogni 3 secondi e stampa lo stato del magazzino per mezzo del metodo print.

```
>java Simulazione 3
 Content:       [       0       0       0       ]
 Order  of t0: -[       7       8       3       ]
 Supply of t1: +[       7       2       0       ]
 Supply of t2: +[       4       7       2       ]
 Content:       [       11      9       2       ]
 Order  of t3: -[       3       1       1       ]
 Order  of t3   dispatched
 Content:       [       8       8       1       ]
 Supply of t4: +[       3       5       7       ]
 Order  of t0   dispatched
 Content:       [       4       5       5       ]
```

Fig.1 – Esempio di funzionamento

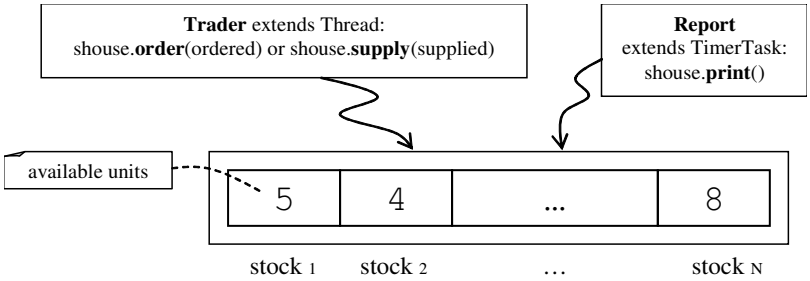


Fig.2 – Struttura dell'oggetto `shouse` di classe `Stockhouse`

TRACCIA PROPOSTA AL CANDIDATO:

```
// StockSet.java
public class StockSet {

    private int[]               stock;

    public StockSet(int numberOfStocks) {
        stock = new int[numberOfStocks];
        for (int i = 0; i < stock.length; i++)
            stock[i] = 0;
    }

    public String toString() {
        String result = "[\t";
        for (int i = 0; i < stock.length; i++)
            result += stock[i] + "\t";
        return result + "]";
    }

    public boolean geq(StockSet other) throws ArrayIndexOutOfBoundsException {
        if (stock.length!=other.numberOfStocks())
            throw new ArrayIndexOutOfBoundsException();

        for (int i = 0; i < stock.length; i++)
            if (stock[i] < other.stock[i])
                return false;
        return true;
    }

    public void order(StockSet ordered) throws ArrayIndexOutOfBoundsException {
        if (stock.length!=ordered.numberOfStocks())
            throw new ArrayIndexOutOfBoundsException();

        for (int i = 0; i < stock.length; i++)
            stock[i] -= ordered.stock[i];
    }

    public void supply(StockSet supplied) throws ArrayIndexOutOfBoundsException {
        if (stock.length!=supplied.numberOfStocks())
            throw new ArrayIndexOutOfBoundsException();

        for (int i = 0; i < stock.length; i++)
            stock[i] += supplied.stock[i];
    }

    public void initRandom(int maxItems) {
        for (int i = 0; i < stock.length; i++)
```

```
            stock[i] = (int)(Math.random()*maxItems);
    }

    public int numberOfStocks() {
        return stock.length;
    }
}

// Stockhouse.java
public class Stockhouse extends StockSet {

    // private part
    private static final int    DEFAULT_NUM_STOCKS = 5;

    public Stockhouse() {
        // …
    }

    public Stockhouse(int numberOfStocks) {
        // …
    }

    public synchronized void supply(StockSet supplied) {
        // …
    }

    public synchronized void order(StockSet ordered) {
        // …
    }

    public synchronized void print() {
        // …
    }

    public synchronized int numberOfStocks() {
        // …
    }
}


// Trader.java
public class Trader extends Thread {

    private Stockhouse house;

    public Trader(String name, Stockhouse house) {
        super(name);
        this.house = house;
    }

    public void run() {
        StockSet stock = new StockSet(house.numberOfStocks());
        stock.initRandom(10);

        if (Math.random()>= 0.5) {
            System.out.println (" Order  of " + getName() + ": -" + stock);
            house.order(stock);
            System.out.println (" Order  of " +  getName() + "   dispatched" );
        }
        else {
            System.out.println (" Supply of " + getName() + ": +" + stock);
            house.supply(stock);
        }
    }
}

// Report.java
import java.util.TimerTask;

public class Report extends TimerTask {
  // …
}

// Simulazione.java
import java.util.Timer;

public class Simulazione {

    public static void attesaRandom() {
```

```
        try {
            Thread.sleep(500 + (long)(Math.random()*2500));
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        final int N = Integer.parseInt(args[0]);

        Stockhouse house = new Stockhouse(N);

        // qui generare il TimerTask  Report e schedularlo

        for (int i=0; i<5; i++) {
            new Trader("t" + i, house).start();
            attesaRandom();
        }
    }
}
```

ESERCITAZIONE TIGA: Thread, TimerTask, … (compito 25/06/04)– SOLUZIONE PROPOSTA

I thread sono singoli flussi di controllo sequenziali, generati all'interno di un processo[28], che permettono di isolare singoli compiti ed eseguirli potenzialmente in parallelo. In Java ad ogni oggetto[29] è implicitamente associato un *lock*, cioè una variabile che permette di determinare se l'oggetto è attualmente acceduto da un altro thread oppure è libero. È possibile controllare l'accesso concorrente di più thread ad uno stesso oggetto dichiarando uno o più metodi o blocchi di codice dell'oggetto come `synchronized`. Si dice anche che ad un oggetto[30] contenente uno i più metodi `synchronized` è asssociato un *monitor* (Hoare), ossia un meccanismo che governa automaticamente l'acquisizione e il rilascio del lock, in modo da consentire accessi in mutua esclusione. Per fornire ai thread la possibilità di coordinare le loro operazioni, ossia di sospendersi e risvegliarsi a vicenda sulla base di condizioni specificate dal programmatore, vi sono le famiglie di metodi `wait` e `notify`, che rispettivamente consentono ad un thread di sospendersi all'interno del monitor di un oggetto e di risvegliarne altri sospesi.

Quando un thread invoca il metodo `wait` su un oggetto, il thread si sospende e rilascia il monitor associato a tale oggetto[31]. I thread che erano sospesi in attesa di accedere al blocco `synchronized` vengono automaticamente risvegliati. Mentre i thread che si erano sospesi mediante `wait` possono essere risvegliati solo da una `notify` o `notifyAll`, a meno che non si adoperi la `wait(timeout)` che provoca il risveglio anche allo scadere del timeout[32].

Questi metodi dovrebbero essere invocati solo dal thread che ha acquisito il monitor, quindi in un blocco notify, altrimenti generano l'eccezione `IllegalMonitorStateException`.

Si osservi che il metodo `o.wait()` pone il thread corrente nel "wait set" dell'oggetto `o`, rilasciando solo il monitor di tale oggetto; non viene rilasciato il monitor di un eventuale altro oggetto in cui il thread corrente risultasse sincronizzato, e ciò potrebbe causare deadlock se l'unico modo per lanciare una `o.notify` fosse quello di acquisire tale secondo monitor.

Si noti che nel metodo `order` non è necessario eseguire una `notifyAll`. Infatti, dopo che un thread acquirente ha prelevato la propria merce, ha svuotato ulteriormente il magazzino. Pertanto, gli altri i thread acquirente sospesi in attesa di alcuni articoli di un certo tipo di merce, sarebbero immediatamente ri-sospesi se venissero risvegliati. Al contrario, un thread fornitore deve invocare la `notifyAll` poichè alcuni thread acquirente potrebbero espletare la loro richiesta a seguito di un aumento di disponibilità di articoli per una data merce.

[28] Un processo è l'attività di esecuzione di una JVM, che esegue il codice di un intero programma. I thread sono anche detti "processi leggeri".

[29] Ogni istanza di `Object` o di una sua sottoclasse. Infatti i metodi `wait`, `notify`, …, appartengono alla classe `Object`.

[30] È possibile dichiarare `synchronized` anche metodic `static`. Ogni classe ha infatti un suo monitor, distinto da quello degli oggetti istanziati.

[31] Invece il metodo `sleep(timeout)` sospende il thread **senza** rilasciare il monitor, per cui è sconsigliabile eseguirlo in un blocco sincronizzato.

[32] Per ulteriori dettagli vedere i metodi delle classi Object e Thread, nelle JAVA API.

Esercizio n. I:

```
// Stockhouse.java
public class Stockhouse extends StockSet {

    private static final int    DEFAULT_NUM_STOCKS =
5;

    public Stockhouse() {
        super(DEFAULT_NUM_STOCKS);
    }
    public Stockhouse(int numberOfStocks) {
        super(numberOfStocks);
    }
    public synchronized void supply(StockSet
supplied) {
        super.supply(supplied);
        notifyAll();
    }
    public synchronized void order(StockSet ordered)
{
        while (!geq(ordered))
            try {
                wait();
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        super.order(ordered);
   }
   public synchronized void print() {
        System.out.println(" Content:       " +
this);
   }
   public synchronized int numberOfStocks() {
        return super.numberOfStocks();
   }
}
```

Esercizio n. II:

```
// Report.java
import java.util.TimerTask;

public class Report extends TimerTask {

    private Stockhouse house;

    public Report(Stockhouse house) {
        super();
        this.house = house;
    }

    public void run() {
        house.print();
    }
}

// Simulazione.java
// parte che genera e schedula il TimerTask  Report

Timer timer = new Timer();
timer.schedule(new Report(house),0,3000);
```

ESERCITAZIONE TIGA: Thread, pattern della sincronizzazione a barriera (compito di esame del 7/06/04)

Scrivere un programma Java che implementi il servizio di *sincronizzazione a barriera* per un insieme di thread che, dopo aver terminato una prima fase della propria attività, devono attendersi tra di loro prima di poter continuare (Fig.1).

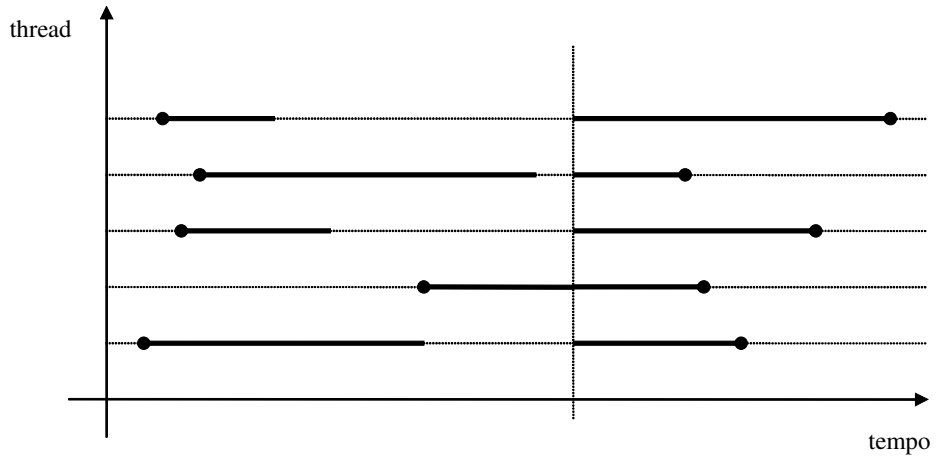


Fig.1 – Sincronizzazione a barriera

Ciascun thread (di classe `RandLifeThread`) attende per un tempo random e poi esegue sulla barriera (di classe `Barriera`) un metodo `waitAll()` che risulta bloccante sino a che è l'ultimo thread ad invocarlo, ed in tal caso esso risveglia tutti i thread.

ESEMPIO DI FUNZIONAMENTO:

```
> java Simulazione 5
 t0 partito...
 t0 sospeso...
 t1 partito...
 t2 partito...
 t3 partito...
 t1 sospeso...
 t4 partito...
 t3 sospeso...
 t2 sospeso...
 t4 sospeso...
 --------------------
 t4 risvegliato...
 t1 risvegliato...
 t2 risvegliato...
 t0 risvegliato...
 t3 risvegliato...
 t1 terminato
 t4 terminato
 t2 terminato
 t0 terminato
 t3 terminato
```

TRACCIA PROPOSTA AL CANDIDATO:

```
// RandLifeThread.java
public class RandLifeThread extends Thread {
       //...
}

// Barriera.java
public class Barriera {
       //...
}

// Simulazione.java
public class Simulazione {
    public static void attesaRandom() {
        try {
            Thread.sleep(500 + (long)(Math.random()*2500));
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        final int N = Integer.parseInt(args[0]);
        Barriera barriera = new Barriera(N);
        for (int i=0; i<N; i++) {
            new RandLifeThread("t" + i, barriera).start();
            attesaRandom();
        }
    }
}
```

ESERCITAZIONE TIGA: Thread, sincr. barriera (compito 7/06/04) – SOLUZ.

La classe `RandLifeThread` contiene un riferimento ad un oggetto `barriera` sul cui monitor l'attività del thread ad un certo punto si sospende (`barriera.waitAll();`). La classe `Barriera` è composta da un semplice contatore, decrementato ad ogni esecuzione di `waitAll()`. Solo a contatore nullo viene eseguita una `notifyAll()` invece della `wait()`, risvegliando tutti i thread.

SOLUZIONE:

```
// RandLifeThread.java
public class RandLifeThread extends Thread {
  private Barriera barriera;
  public RandLifeThread(String name,
                        Barriera barriera) {
      super(name);
      this.barriera = barriera;
  }
  public void run()   {
    System.out.println(getName() + " partito...");
    Simulazione.attesaRandom();
    System.out.println(getName() + " sospeso...");
    barriera.waitAll();
    System.out.println(getName() + "
risvegliato...");
    Simulazione.attesaRandom();
    System.out.println(getName() + " terminato");
  }
}

// Barriera.java
public class Barriera {
  int contaThread;
  public Barriera(int numThread) {
      contaThread = numThread;
  }
  public synchronized void waitAll() {
    if (contaThread > 1) {
      contaThread--;
      try {
        wait();
      }
      catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    else {
      System.out.println(" ------------------");
      notifyAll();
    }
  }
}
```

APPROFONDIMENTO: NESTED MONITOR LOCKOUT.

Precedentemente[33] abbiamo analizzato un esempio di attesa indefinita di alcune attività (*starvation*). Vediamo ora un esempio di blocco reciproco di tutte le attività del sistema (*deadlock*), basato sull'impiego dei monitor innestati[34].

Un problema molto comune, e difficile da individuare, è quello che viene chiamato "nested monitor lockout". Questo si crea quando un monitor object è posto dentro un altro monitor object. Si consideri il seguente codice:

```
// T.java
import java.lang.Thread;

class A {
  private boolean cond = false;

  public synchronized void waitCond() {
    System.out.println(" bloccato...");
      while(!cond)
      try { wait(); }
      catch(Exception e){}
      System.out.println("sbloccato");
  }

  public synchronized void notifyCond(boolean c) {
    cond = c;
    notifyAll();
  }
}

class B {
  protected A a = new A();

  public synchronized void attendiCond() {
    a.waitCond();
  }

  public synchronized void notificaCond(boolean c){
    a.notifyCond(c);
  }
}

public class T extends Thread {

  private static B b = new B();

  public T(String name) {
    super(name);
  }

  public void run() {
    if (getName().equals("t0")) {
      b.attendiCond();                    // (1)
      //b.a.waitCond();                    // (2)
    }
    else {
      b.notificaCond(true);               // (1)
      //b.a.notifyCond(true);              // (2)
    }
  }

  public static void main(String[] args)
                              throws Exception {
      new T("t0").start();
      sleep(1000); // attendi che t0 si sospenda
      new T("t1").start();
  }
}
```

Il thread *t0* esegue *b.attendiCond()* ma, poiché gli oggetti *a* e *b* hanno ciascuno il proprio monitor, la chiamata *wait*() dentro *A.waitCond* rilascia il monitor dell'oggetto *a* ma non quello dell'oggetto *b*. In questo modo il thread *t1* non potrà mai chiamare *b.notificaCond()* per sbloccare *t1*. Quindi entrambi i thread rimangono incessantemente bloccati.

Nella variante in cui le due righe *//(1)* sono sostituite dalle righe *//(2)* il deadlock non avviene, in quanto viene adoperato un unico monitor, quello di *a,*per invocare direttamente i metodi relativi.

[33] Vedere il laboratorio su "TCP server e TCP client..."

[34] Vedere il laboratorio su "Thread, TimerTask...." per una introduzione a tale problema.

Una versione semplificata del nested monitor lockout è la seguente, in cui la classe *S* contiene due blocchi sincronizzati innestati, relativi a due diversi oggetti: *r* (this) e l' oggetto *o*. Togliendo il commento *///(1)* i due oggetti vengono a concidere, per cui è il meccanismo dei *lock rientranti* che evita l'auto-deadlock.

```
// S.java
import java.lang.Thread;
class R {
    private static Object o = new Object();

    public synchronized void aspetta() {
        System.out.println( " bloccato ");
        //o = this;                    // (1)
        synchronized(o) {
                try { o.wait(); }
                catch (Exception e) {}
        }
        System.out.println(" sbloccato ");
    }
    public synchronized void notifica() {
        synchronized(o) {
                o.notifyAll();
        }
    }
}
public class S extends Thread {
    private static R r = new R();

    public S(String name) {
        super(name);
    }
    public void run() {
       if (getName().equals("t0"))
                r.aspetta();
        else
                r.notifica();
    }
    public static void main(String[] args)
                                throws Exception {
      new S("t0").start();
      sleep(1000); // attendi che t0 si sospenda
      new S("t1").start();
    }
}
```

Esiste anche il ***livelock***, una situazione di stallo in cui più attività cambiano continuamente il proprio stato, in reciproca risposta, senza eseguire elaborazioni utili. È simile al deadlock nel senso che non avvengono progressi nell' avanzamento delle attività (al massimo si oscilla tra alcuni stati intermedi) ma differisce per il fatto che nessuna attività è sospesa in attesa di un'altra e quindi viene impegnato attivamente il processore.

Un esempio umano di livelock avviene quando due persone si incontrano in un corridoio, o davanti ad un passo stretto. Ciascuna cerca di mettersi di lato per lasciare il passo all'altra, ma si spostano dallo stesso lato e approssimativamente allo stesso tempo, per cui finiscono per dondolare assieme da una parte all'altra ostacolandosi reciprocamente.

Un altro esempio di livelock è quando, l'ultimo giorno utile per il pagamento delle imposte, tutti gli utenti accedono ripetutamente al server del ministero delle finanze saturando le risorse del sistema e rendendolo inutilizzabile, oppure quando due persone tentano di telefonarsi a vicenda trovando occupato.

Una soluzione per il livelock è cambiare il comportamento dopo un certo timeout, oppure rieseguire la medesima attività ad intervalli random più lunghi. Ad esempio, nel laboratorio su TCP client e server, nella versione senza uso dei thread, avevamo progettato dei client "intelligenti", che inviano richieste ad intervalli random[35] e, in caso di rifiuto di connessione al server, aumentano l'attesa prima della prossima richiesta, terminando del tutto dopo tre tentativi.

Un altro problema tipico della programmazione concorrente è quello della ***race condition (corsa critica)***, che può essere definita come un comportamento anomalo dovuto alla dipendenza critica inattesa della sincronizzazione relativa ad eventi. Le race condition coinvolgono generalmente uno o più processi che accedono ad una risorsa comune, dove questo accesso multiplo non è stato gestito correttamente. Tipicamente sono casi difficili da individuare mediante debug, perché il programmatore ha supposto in modo errato che un evento particolare accadrà sempre prima di un altro, senza contemplare la possibilità che le attività possano essere ritardate o invertite dallo scheduler del sistema operativo.

Ad esempio, nel codice relativo a *T.java*, su una macchina monoprocessore l'istruzione *sleep(1000)* può essere omessa senza produrre alcun cambiamento nella esecuzione dell'applicazione, in quanto i due thread vengono eseguiti uno dopo l'altro. In un sistema biprocessore[36] i due thread verranno eseguiti quasi in parallelo, per cui è possibile che il secondo thread arrivi a modificare la condizione a *true* prima che il primo thread si sia bloccato, e questo non riprodurrebbe il deadlock, cioè il risultato previsto.

[35] Un periodo di richiesta con un margine di casualità permette di distribuire il carico di richieste ed evitare, in una situazione di congestione del traffico, che questa si ripeta esattamente dopo qualche istante a causa del medesimo periodo. I protocolli di rete adoperano una simile strategia.

[36] I processori *Pentium IV HT* dell'aula didattica supportano la tecnologia *HyperThreading*, in cui il sistema operativo "vede" due CPU logiche, e l'architettura è in grado di supportare un certo livello di parallelismo. Senza l'istruzione *sleep(1000)* è semplice, dopo una decina di tentativi, riprodurre la condizione di "inversione dei thread" in cui non si ha deadlock. È anche possibile disattivare l'hyperthreading modificando la proprietà *Affinity* del processo *java.exe*, nel seguente modo. Dal *Windows Task Manager* (CTRL+SHIFT+ESC), dopo aver lanciato il processo *java.exe*, selezionare la scheda *processes* e poi il processo *java.exe* con il tasto destro, quindi selezionare *set Affinity* e disabilitare una CPU. in tal modo si delega una unica CPU al processo java (e quindi ai suoi thread). Per poter avere il tempo di modificare la *Affinity* sul processo *java.exe* in esecuzione, modificare il codice del metodo *main*, inserendo prima del lancio dei thread un'attesa di 15 secondi(*sleep(15000))*.

ESERCITAZIONE TIGA: RMI, unico host, caricamento locale delle classi

Un `Conto` bancario è caratterizzato esclusivamente da un `nome` ed un `saldo`. Il `nome` identifica univocamente un `Conto`, tra gli oggetti istanziati. Un oggetto di classe `Conto` viene implementato come un *oggetto remoto*, ossia con (alcuni) metodi invocabili da applicazioni in esecuzione su altre JVM (inizialmente supposte, per semplicità, residenti su un unico host) tramite l'*interfaccia remota* `Conto` (Fig.1):

```
public interface Conto extends Remote {
  void versa(double importo)    throws RemoteException;
  void preleva(double importo   throws RemoteException,
                               NegativeAmountException;
  double ritornaSaldo()         throws RemoteException;
}
```

Nella creazione dell'oggetto `ContoImpl`, si definisce il nome dell'oggetto e si imposta il saldo iniziale al valore zero. Il nome del conto viene anche adoperato come *nome pubblico* per registrare l'oggetto *Conto* presso l'RMI Registry.

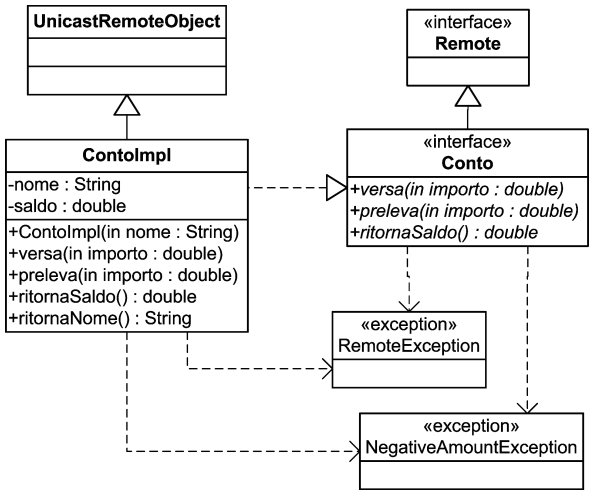


Fig. 1 – Diagramma delle Classi.

PROGETTARE E REALIZZARE:

• La classe `ContoImpl`, implementazione **thread-safe** dell' interfaccia. A tale scopo, si dichiarino semplicemente i metodi come `synchronized` per garantire un accesso mutuamente esclusivo.

• La classe `ContoServer`, che implementa il programma *server* in accordo alle seguenti specifiche:
  i) *crea* (ed *esporta*) un insieme di conti, dai nomi passati da riga di comando;
  ii) li *registra* presso l'RMI Registry.

• La classe `ContoClient`, che implementa il programma *client* in accordo alle seguenti specifiche:
  i) riceve *`nome`*, *`importo1`* ed *`importo2`* come parametri di ingresso, da riga di comando;
  ii) recupera un oggetto di classe `ContoImpl_Stub` (che funge da proxy per l'oggetto `ContoImpl` residente sul server) con tale *`nome`*, e lo riferisce con un riferimento interfaccia `conto`.
  iii) invoca remotamente i seguenti metodi:

```
conto.versa(importo1); conto.preleva(importo2);
System.out.println(conto.ritornaSaldo());
```

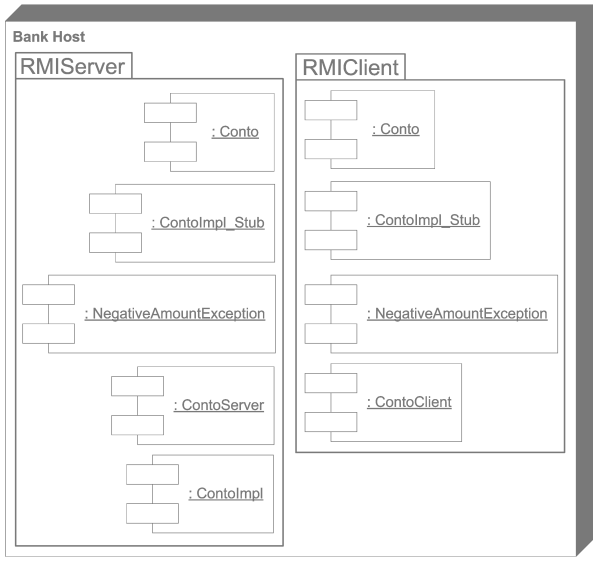


Fig. 2 – Diagramma di distribuzione dei componenti.

• La gestione delle seguenti eccezioni: `NotBoundException` e `java.net.MalformedURLException` (metodo `Naming.`**`lookup`**), `RemoteException` (metodi dell' interfaccia `Conto`), `AlreadyBound Exception` (metodo `Naming.`**`bind`**), `NegativeAmountException` (classe eccezione sviluppata "ad hoc" per intercettare un prelievo superiore al saldo, quindi sollevata dal metodo `Conto.`**`preleva`**).

SUGGERIMENTI:

- Compilare tutti i sorgenti (`javac *.java`) in un' unica cartella, successivamente produrre la classe *stub* mediante l'applicazione *rmic* della directory *bin* (`rmic -v1.2 ContoImpl`), e distribuire il bytecode in tre cartelle, *RMIregistry, RMIserver* ed *RMIclient*, come indicato in Fig.1-a e Tab.1.

Tab. 1 – Moduli necessari ad ogni sottosistema, a compile e run time.

	ContoServer	ContoClient	RMI Registry
compile time	server, interfaccia remota e relativa implementazione, altre classi adoperate (da questi moduli).	client, interfaccia remota, altre classi adoperate (da questi moduli).	
run time	server, interfaccia remota e relativa implementazione, altre classi adoperate, stub	client, interfaccia remota, altre classi adoperate, stub.	interfaccia, altre classi adoperate, stub.

- Adoperare l'applicazione *rmiregistry* della directory *bin*, digitando `rmiregistry 1099` [37] per far partire l'RMI Registry, prima del client e del server, dalla cartella *RMIregistry*.
- Avviare i processi `ContoServer` e `ContoClient` dai rispettivi percorsi (quindi su JVM differenti).

[37] Se vi sono problemi di connettività con RMIRegistry, provare a cambiare porta (es. `1100`, `1101`,...).

- In `ContoClient` passare come parametro di ingresso anche l'host di residenza del registry, in modo da costruire, per il metodo `lookup`, una URL come nel seguente esempio `//10.114.109.11:1099/pippo`[38].

ESERCITAZIONE TIGA: RMI, caricamento locale delle classi – SOLUZIONE PROPOSTA

Progettazione:

In Fig.1 viene mostrata la distribuzione dei file. Si osservi che il client contiene solo l'interfaccia remota `Conto` e la classe `_stub`[39], ossia il proxy che funge da riferimento all'oggetto remoto (`ContoImpl`, residente sul server) per tutte le invocazioni ad esso destinate. L'RMI Registry é sostanzialmente un servizio di naming non persistente che consente al server di pubblicare i servizi e al client di recuperarne il proxy.

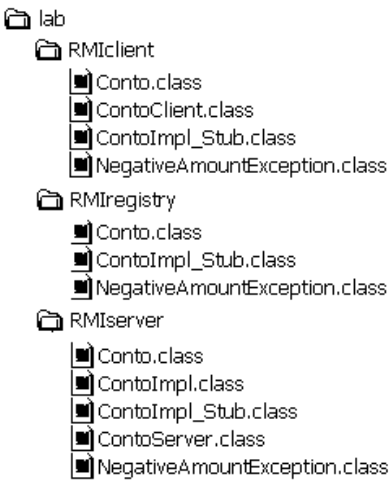


Fig.1 – distribuzione dei files

Si noti che il proxy (stub) ritornato dal metodo `lookup` in `ContoClient` viene gestito, tramite il riferimento interfaccia *conto*, allo stesso modo in cui sarebbe gestito il corrispondente oggetto remoto, qualora venisse riferito localmente dal un riferimento interfaccia *conto*. Ma è necessario evidenziare una grossa differenza. Quando il client invoca un metodo remoto passando come parametro un riferimento ad oggetto, viene effettuata una copia dell'oggetto[40] dal client al server, e lo stesso succede dal server al client quando viene ritornato il risultato. Mentre nella invocazione locale si passa semplicemente un riferimento all'oggetto, e se un metodo modifica l'oggetto passato, questa modifica si ripercuoterà sull'oggetto chiamante che al ritorno troverà l'oggetto modificato. Invece nell'invocazione di metodi remoti, il server ottiene una copia dell'oggetto che è indipendente dal quella riferita come parametro, quindi modificare un oggetto non è sufficiente per ritornare un risultato: bisogna anche ritornarlo come risultato per fare sì che il chiamante veda le modifiche.

Questo ha anche un impatto sull'efficienza. Non si può passare "a cuor leggero" come parametro un oggetto complesso, altrimenti si potrebbe copiare involontariamente una enorme quantità di dati (e in Java questo può succedere più facilmente di quanto non sembri). Poichè la copia di oggetti in andata (passaggio parametri) e ritorno (valori risultanti) avviene utilizzando il meccanismo della serializzazione. Diverso è il comportamento se il riferimento è ad un oggetto remoto. In tal caso viene passato un riferimento remoto, cioè uno stub dell'oggetto[41].

Il registry va avviato su ciascun host dove risiedono servizi di oggetti remoti e si accettano richieste di servizio. La porta 1099 é di default, e può anche essere omessa.

Per riusare un nome (già presente nel registry) per un nuovo oggetto remoto, adoperare il metodo:

```
public static void Naming.rebind(String objectname,Remote obj)
```

Mentre per eliminare semplicemente l'associazione con l'oggetto remoto:

```
public static void Naming.unbind(String objectname)
```

Infine, per avere un elenco dei nomi registrati:

```
public static String[] list(String registryname)
```

dove `registryname` puó anche essere il percorso `//host:porta/` definito da una URL senza alcun nome finale.

Codifica in linguaggio Java:

```
// Conto.java

import java.rmi.*;

public interface Conto extends Remote {
```

---

[38] Cerca l'oggetto `pippo` nel registry che risiede in `10.114.109.11` sulla porta di default `1099`).

[39] Letteralmente *stub* significa "surrogato", ed implementa il design pattern **proxy** in quanto "fa credere" al client di avere a disposizione il componente originale (l'implementazione residente sul server). Si esegua il seguente esperimento. Decompilare la classe "*_stub*" mediante il *Java Decompiler* ed osservare a) che essa estende *RemoteStub*, una superclasse comune per gli stub; b) implementa l'interfaccia *Remote*, priva di metodi perchè funge da semplice marcatore, similmente a *Serializable*; c) implementa l'interfaccia *Conto*, ma solo nominalmente, in quanto serializza i parametri delle chiamate e si occupa della invocazione dei metodi destinati all'oggetto remoto e del recupero dei risultati. Il fatto che la classe stub "implementi" *Conto* permette l'assegnamento del riferimento interfaccia *conto* ad un oggetto stub. Per verificare che l'oggetto ritornato dalla lookup sia solo uno stub, provare a scrivere:
`Object o = Naming.lookup(...); System.out.println(o.getClass().getName());` per ottenere "`ContoImpl_Stub`".

[40] Che pertanto deve essere serializzabile. Il passaggio di oggetti primitivi aviene per valore, analogamente all'invocazione locale di metodo.

[41] Ad esempio, il riferimento di un *callback object*: un riferimento remoto di un oggetto remoto lato client, passato al server tramite un metodo di registrazione invocato su un oggetto remoto lato server. Il meccaniso è il seguente 1) Il client recupera attraverso il registry il riferimento all'oggetto remoto *S* del server; 2) il client esegue l'invocazione *S.registra(C)*; 3) nel corpo di tale metodo, il server recupera il riferimento remoto all'oggetto remoto *C*; 4) quindi con tale riferimento può eseguire notifiche al client tramite un metodo remoto di *C*.

```
    void       versa(double importo)         throws RemoteException;
    void       preleva(double importo)       throws RemoteException, NegativeAmountException;
    double     ritornaSaldo()                throws RemoteException;
}

// ContoImpl.java

import java.rmi.*;
import java.rmi.server.*;

public class ContoImpl  extends UnicastRemoteObject implements Conto {

    public ContoImpl(String n) throws RemoteException {
        nome  = n;
        saldo = 0.0;
    }

    public synchronized void preleva(double importo) throws RemoteException, NegativeAmountException {
        if (saldo < importo)
            throw new NegativeAmountException();
        else
            saldo -= importo;
    }

    public synchronized void versa(double importo) throws RemoteException {
        saldo += importo;
    }

    public synchronized double ritornaSaldo() throws RemoteException {
        return saldo;
    }

    public synchronized String ritornaNome() {
        return nome;
    }

    private String nome;
    private double saldo;

}

// NegativeAmountException.java

public class NegativeAmountException extends Exception {
    public NegativeAmountException() {}
    public NegativeAmountException(String message) {
        super(message);
    }
}

// ContoServer.java

import java.rmi.*;

public class ContoServer {

    public static void main(String[] args) {
        String nome = null;
        try {
            // creo ed esporto gli oggetti
            ContoImpl[] conto = new ContoImpl[args.length];
            for (int i = 0; i < args.length; i++)
                conto[i] = new ContoImpl(args[i]);

            // li registro presso il registry
            System.out.println(MSG_USR1);

            for (int i = 0; i < args.length; i++) {
                nome = conto[i].ritornaNome();
                Naming.bind("//localhost:1099/" + nome, conto[i]);
                System.out.println(nome);
            }
        }
        catch (RemoteException e) {
                System.err.println(MSG_ERR3);
                System.exit(1);
        }
        catch (AlreadyBoundException e) {
```

```
                System.err.println(nome + MSG_ERR4);
        }
         catch(java.net.MalformedURLException e) {
                System.err.println(MSG_ERR5);
        }
    }
    private static final String MSG_USR1 = "Attesa delle invocazioni remote di metodi per gli oggetti: ";

    private static final String MSG_ERR3 = " Impossibile contattare il registry.";
    private static final String MSG_ERR4 = " e' un nome gia' associato ad un oggetto nel registry.";
    public  static final String MSG_ERR5 = " URL errata. ";
}

// ContoClient.java

import java.rmi.*;

public class ContoClient {

    public static void main(String[] args) {

        final String URL    = "//" + args[0] + ":1099/";
        final String NOME   = args[1];

        try {
            // ricerca oggetto remoto nel registro, per nome
            Conto conto = (Conto)Naming.lookup(URL + NOME);

            conto.versa(Double.parseDouble(args[2]));
            conto.preleva(Double.parseDouble(args[3]));
            System.out.println("Nuovo saldo di " + NOME + ": " + conto.ritornaSaldo());

            System.out.println("Elenco nomi registrati:");
            String[] elenco = Naming.list(URL);
            for (int i=0; i<elenco.length; i++)
                System.out.println(elenco[i]);
        }
        catch (NotBoundException e) {
                System.err.println(NOME + MSG_ERR1);
        }
        catch (RemoteException e) {
                System.err.println(MSG_ERR2 + NOME + ".");
        }
        catch(NegativeAmountException e) {
                System.err.println(MSG_ERR4);
        }
        catch(java.net.MalformedURLException e) {
                System.err.println(ContoServer.MSG_ERR5);
        }
    }

    private static final String MSG_ERR1 = " e' un nome non associato ad alcun oggetto nel registry.";
    private static final String MSG_ERR2 = " Impossibile individuare l'oggetto remoto ";
    private static final String MSG_ERR4 = " Impossibile prelevare il quantitativo indicato. ";
}
```

Testing del programma:

• In Fig.2 vengono sollevate alcune delle eccezioni gestite: assenza del registry (1), nome non registrato (3), versamento di € 100 e prelievo di € 40 (5), prelievo di un importo maggiore al saldo (6), quest'ultimo evento identificato dall'eccezione `NegativeAmountException`. In Fig.3 viene mostrato il comportamento di client e server senza l'attivazione del registry.

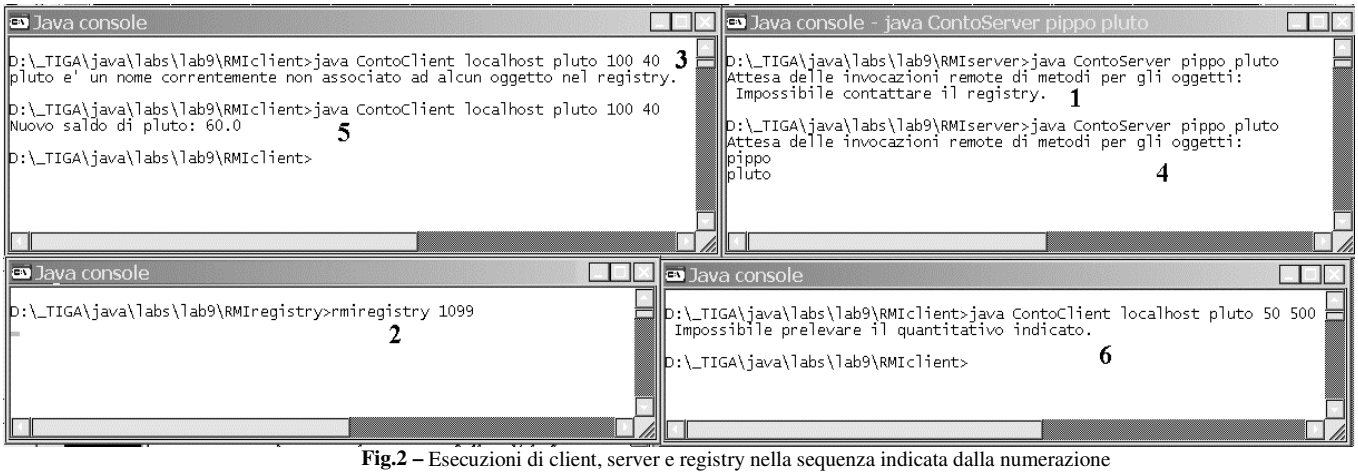


Fig.2 – Esecuzioni di client, server e registry nella sequenza indicata dalla numerazione

• Infine la Fig.4 mostra l'utilizzo normale dei servizi: l'avvio del registro (1), la registrazione di due nomi (2), deposito e prelievo da diversi processi (3-5) , utilizzo di un nome non associato ad alcun oggetto remoto (6).

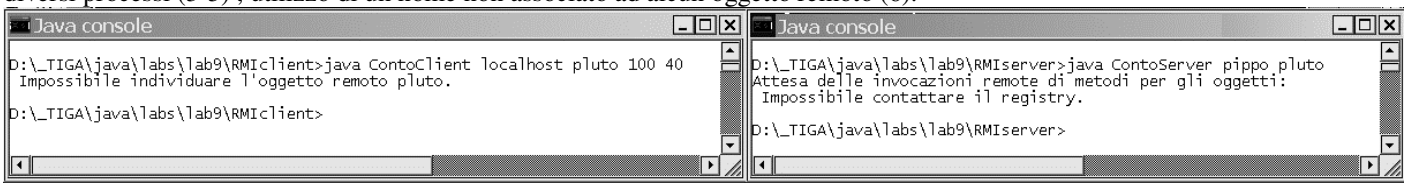


Fig.3 – Esecuzione di client e server senza registry

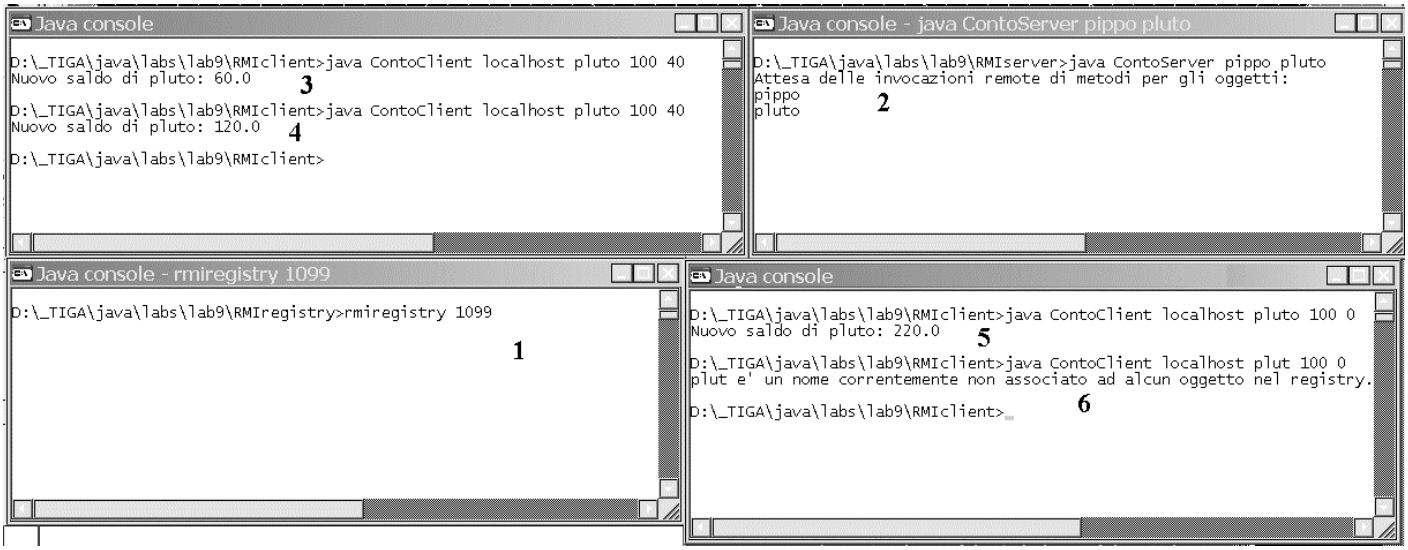


Fig.4 – Scenario di esecuzione standard, nella sequenza indicata dalla numerazione.

Complementi:

• Invece di invocare l'RMIregistry con un processo indipendente, mediante il programma `rmiregistry` della Sun, il programmatore può creare all'interno del codice un proprio registry (classe `java.rmi.registry.LocateRegistry`),

```
public static Registry createRegistry(int port)
```

che verrà eseguito come demone, dal comportamento personalizzabile, nella medesima istanza della JVM.

• È possibile anche accedere a registry remoti già lanciati, mediante:

```
Registry getRegistry(String host, int port)
```

dove `host` e `port` sono opzionali ed assumono – di default – i valori `localhost` e `1099` rispettivamente.

• Accedendo al registry (individuabile interrogando tutte le porte di un host) sarebbe possibile ridirigere per scopi maliziosi le chiamate ai server RMI registrati (es. `list()` + `rebind()`). Per questo motivo, i metodi `bind()`, `rebind()` e `unbind()` sono invocabili solo dall'host su cui è in esecuzione il registry, ossia non è possibile effettuare modifiche della struttura client/server dall'esterno.

• Il seguente script automatizza parte delle procedure viste sinora. Quali risultati produce ?

```
rem make.bat
set CLASSPATH=
javac *.java
rmic -v1.2 ContoImpl
copy /Y Conto.class .\RMIclient
copy /Y ContoImpl_Stub.class .\RMIclient
copy /Y NegativeAmountException.class .\RMIclient
copy /Y ContoClient.class .\RMIclient
copy /Y Conto.class .\RMIregistry
copy /Y ContoImpl_Stub.class .\RMIregistry
copy /Y NegativeAmountException.class .\RMIregistry
copy /Y *.class .\RMIserver
del  RMIserver\ContoClient.class
pause
start cmd /k "color 0F && rmiregistry 1099"
pause
start cmd /k "color 1F && java ContoServer pippo pluto"
pause
start cmd /k "color 2F && java ContoClient localhost pluto 100 40"
start cmd /k "color 3F && java ContoClient localhost plut 100 40"
```

- Eseguire dei processi Client su un host diverso dal server.

ESERCITAZIONE TIGA: RMI, multi-host, caricamento remoto delle classi

Relativamente alla esercitazione su "RMI,…,caricamento locale delle classi", si realizzi il medesimo servizio di conto corrente remoto, con le seguenti modifiche.

- `ContoClient` e `ContoServer` sono residenti su due host diversi (Fig,1), nelle rispettive cartelle `rmiclient` e `rmiserver`. Per motivi di sicurezza, il server ha il registry avviato in *locale*, dalla cartella `rmiregistry`, in ascolto sulla porta `1110` [42].
- `ContoClient` non riceve più come parametro di ingresso il nome dell'oggetto, ma sceglie a caso un nome all'interno della lista di nomi ottenuta dal registry invocando il metodo:

```
String[] elenco = Naming.list(URL)
```

- Le classi `Conto`, `ContoImpl_stub` e `NegativeAmountException` sono disponibili sul seguente web server:

```
http://www2.ing.unipi.it/~o1553499/rmiclasslocation/
```

  quindi scaricate dinamicamente da client e registry all'occorrenza.

- I file sorgente di client e server sono compilati sui rispettivi host, per cui occorre adattare il codice di `ContoClient` in modo che questo sia possibile.

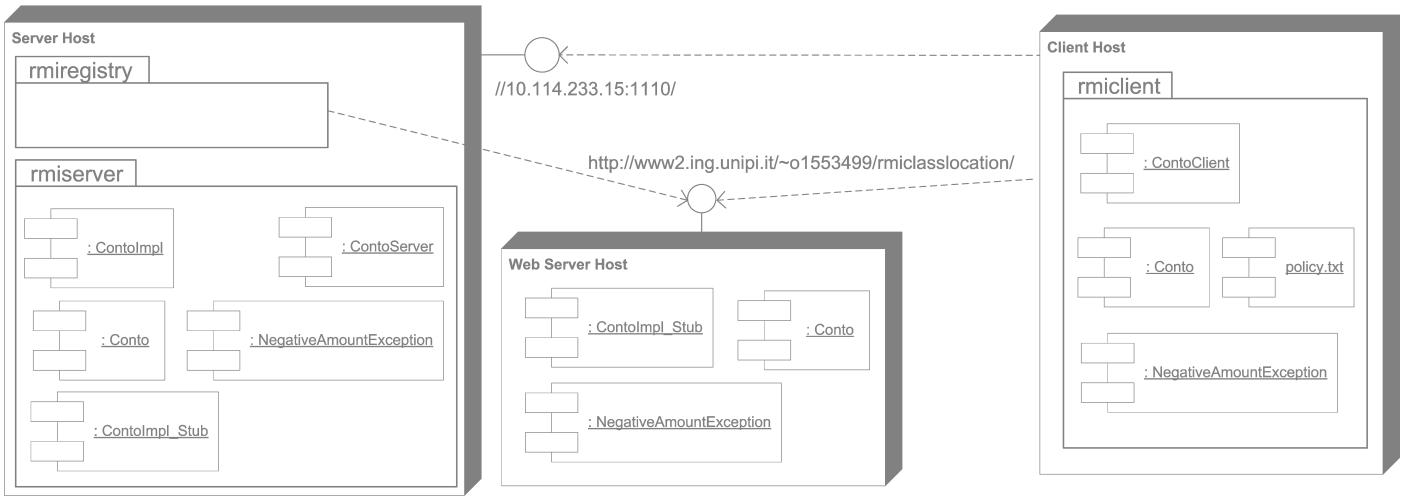


Fig. 1 – Diagramma di distribuzione dei componenti

PROGETTARE E REALIZZARE:

- I moduli residenti su Server Host e Client Host, specificando l'URL del web server di Fig.1 come codebase, e le seguenti proprietà di policy:

```
// policy.txt
grant {
    permission java.security.AllPermission;
};
```

- Adoperare un altro web server, avviato su un host del laboratorio [43] .

SUGGERIMENTI:

- Prima di compilare o eseguire, azzerare il CLASSPATH.
- Digitare `rmiregistry 1110` per far partire l'RMI registry, prima del client e del server, dalla cartella *rmiregistry*.
- Eseguire l'installazione di *Tomcat* in modalità "full", indicando il percorso che contiene la cartella *bin* del JSDK (senza tale cartella, es. *C:\j2sdk1.4.2_06*). Uccidere eventuali processi *tomcat5.exe* in esecuzione, ed avviare il web server mediante lo script *bin\startup.bat*, controllando che nella console non vi siano eccezioni[44]. Eseguire un test funzionale accedendo mediante un browser alla URL *http://localhost:8080* in locale, e poi[45] *http:// 10.114.233.15:8080* in remoto. Quindi copiare la cartella *rmiclasslocation* in *C:\Program Files\Apache Software Foundation\Tomcat 5.0\webapps\ROOT* e poi accedere con un web browser ad *http://10.114.233.15:8080/rmiclasslocation*. Dovrebbe comparire l'elenco dei file *.class*.

---

[42] Infatti, eventuali `bind`, `unbind`, `rebind` eseguite da un server su un registry remoto produrrebbero una `AccessException`. Questo per impedire che un utente malintenzionato possa danneggiare il servizio registrando nomi di oggetti remoti o effettuando un overriding delle interfacce remote.

[43] Si usi *Apache Tomcat* (http://jakarta.apache.org/site/downloads/downloads_tomcat-5.cgi).

[44] In caso di eccezioni terminare l'applicazione e riavviarla. Per terminare *Tomcat*, usare sempre lo script *bin\shutdown.bat*.

[45] Ovviamente l'indirizzo IP e` di esempio.

- Avviare i processi `ContoServer` e `ContoClient` dai rispettivi percorsi, secondo il seguente esempio:

```
java -Djava.rmi.server.codebase= http://www2.ing.unipi.it/~o1553499/rmiclasslocation/ ContoServer mio tuo
java -Djava.security.policy=policy.txt ContoClient 10.114.233.15 100 50
```

ESERCITAZIONE TIGA: RMI, caricamento remoto delle classi – SOLUZIONE PROPOSTA

Progettazione:

In Fig.1 viene mostrata la distribuzione dei file nelle varie cartelle. Lo script *make.bat* genera e distribuisce tutti i componenti *.class* adoperando i file *.java* presenti nelle cartelle *javaclient* e *javaserver*, nel caso in cui tutto risieda su un unico host. Si noti che il registry non ha la visibilità diretta di alcun file `.class`. Gli script *go.bat* azzerano il classpath e lanciano i vari processi.

Il codebase deve essere specificato solo dal server, in quanto é una proprietà che viene annotata nel riferimento remoto (stub) pubblicato sul registry; questi sarà il primo ad adoperare il codebase quando il server invocherà una `bind` quindi occorrerà caricare il file `ContoImpl_Stub.class` e crearne una istanza da legare al nome.

Quando il client invocherà una `lookup` sul registry, questi ritornerà una istanza di `ContoImpl_Stub.class` (non il file `ContoImpl_Stub.class`), relativa all'oggetto remoto registrato; questo oggetto serializzato conterrà anche informazioni sul codebase. Quindi anche il client saprà da dove scaricare il file `ContoImpl_Stub.class` per poter deserializzare l'istanza[46]

Al fine di permettere anche al registry il caricamento delle classi dal codebase é importante che non sia settata alcuna variabile ambiente `CLASSPATH`.

Il Server ha già tutte le classi che gli occorrono e pertanto non caricherá bytecode, e comunque – anche se dal medesimo host sarà il registry a caricarle – questo é un host "di fiducia" che é proprio il server ad indicare nel suo codebase.

Al contrario, il client carica del codice (`ContoImpl_Stub.class`) da un server che non conosce direttamente (l'interfaccia é una sorta di dichiarazione di intenti, non si sa cosa realmente facciano i metodi della classe "*_stub*", ad es. potrebbero accedere al file system locale) quindi é opportuno che esso specifichi una policy per proteggersi, costruendo un *SecurityManager* e personalizzandolo mediante il file *policy.txt*.

Il caricamento remoto dello stub permette al server di modificare l'implementazione dei servizi e rigenerare la classe stub senza notificarlo ai client. Il codebase può anche essere adoperato per caricare classi diverse dallo stub. Nella piattaforma *Java SDK 1.5.0* esiste anche il supporto per generare dinamicamente la classe dello stub a tempo di esecuzione, senza adoperare *rmic*[47]. L'evoluzione di RMI tende a fornire al programmatore un supporto sempre più trasparente per realizzare applicativi distribuiti, consentendo la distribuzione a livello di oggetto e permettendo al client di interagire con esso come fosse disponibile in locale, nascondendo completamente i dettagli della comunicazione.

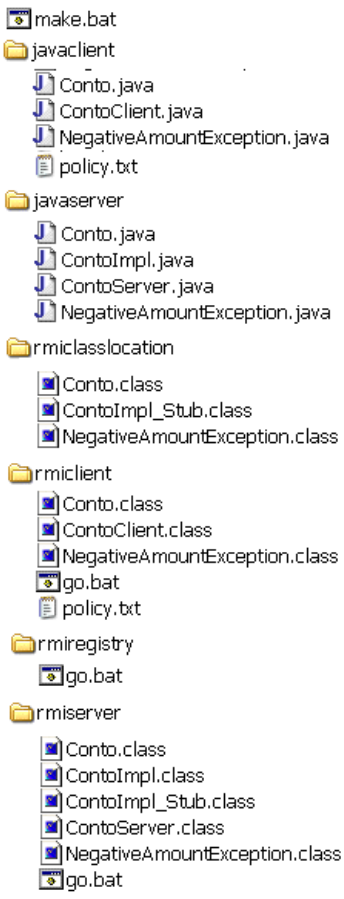


Fig.1 – Distribuzione dei file

Ad esempio, tale vantaggio è evidente con i web browser e le applet. Tramite RMI si possono spostare sul server molte operazioni normalmente svolte dal client. Con le applet che accedono ad un database, invece di caricare sul client un driver JDBC, si può incapsulare la gestione del database in un server RMI che gira ad esempio sul medesimo host di residenza del DBMS.

Le modifiche al codice, che riportiamo integralmente per completezza, riguardano il client: la creazione del security manager, la richiesta della lista di nomi e la scelta casuale di uno di essi, la duplicazione della stringa `MSG_ERR5` che non può più essere condivisa con il server.

Codifica in linguaggio Java:

```
// Conto.java

import java.rmi.*;

public interface Conto extends Remote {
```

[46] Nota bene: il riferimento pubblicato è ad un oggetto non ad una classe, quindi vengono serializzate solo le informazioni che ne caratterizzano l'istanza, pertanto nessuna informazione sui metodi, sulle costanti, sulle variabili static o transient. Al momento della deserializzazione sarà ricreata una copia dell'istanza "trasmessa" usando il file `ContoImpl_Stub.class` (che deve quindi essere accessibile) e le informazioni di istanza ricevute.

[47] In tal caso l'oggetto stub è una istanza di java.lang.reflect.Proxy (la cui classe si può generare dinamicamente) con un java.rmi.server.RemoteObject InvocationHandler come gestore di invocazione. Questo comportamento si può imporre impostando la proprietà di sistema java.rmi.server.ignoreStubClasses ="true". Per ulteriori dettagli, http://java.sun.com/j2se/1.5.0/docs/guide/rmi/relnotes.html.

```
    void       versa(double importo)         throws RemoteException;
    void       preleva(double importo)       throws RemoteException, NegativeAmountException;
    double     ritornaSaldo()                throws RemoteException;
  }


// ContoClient.java

import java.rmi.*;

public class ContoClient {

    public static void main(String[] args) {

        final String URL  = "//" + args[0] + ":1110/";
              String nome = null;

        if (System.getSecurityManager() == null) {
            System.setSecurityManager( new RMISecurityManager() );
        }
        try {
            // ricerca oggetto remoto nel registro, per nome

            System.out.println("Elenco nomi registrati:");
            String[] elenco = Naming.list(URL);
            for (int i=0; i<elenco.length; i++)
                System.out.println(elenco[i]);

            nome = elenco[(int)Math.floor(Math.random()*elenco.length)];

            Conto conto = (Conto)Naming.lookup(nome);

            conto.versa(Double.parseDouble(args[1]));
            conto.preleva(Double.parseDouble(args[2]));
            System.out.println("Nuovo saldo di " + nome + ": " + conto.ritornaSaldo());
        }
        catch (NotBoundException e) {
                System.err.println(nome + MSG_ERR1);
        }
        catch (RemoteException e) {
                System.err.println(MSG_ERR2 + nome + ".");
                 e.printStackTrace();
        }
        catch(NegativeAmountException e) {
                System.err.println(MSG_ERR4);
        }
        catch(java.net.MalformedURLException e) {
                System.err.println(MSG_ERR5);
        }
    }

    private static final String MSG_ERR1 = "e' un nome non associato ad alcun oggetto nel registry.";
    private static final String MSG_ERR2 = " Impossibile individuare l'oggetto remoto ";
    private static final String MSG_ERR4 = " Impossibile prelevare il quantitativo indicato. ";
    private static final String MSG_ERR5 = " URL errata. ";

}

// ContoImpl.java

import java.rmi.*;
import java.rmi.server.*;

public class ContoImpl  extends UnicastRemoteObject
                        implements Conto {

    public ContoImpl(String n) throws RemoteException {
        nome  = n;
        saldo = 0.0;
    }

    public synchronized void preleva(double importo) throws RemoteException, NegativeAmountException {
        if (saldo < importo)
            throw new NegativeAmountException();
        else
            saldo -= importo;
    }
```

```
    public synchronized void versa(double importo) throws RemoteException {
        saldo += importo;
    }

    public synchronized double ritornaSaldo() throws RemoteException {
        return saldo;
    }

    public synchronized String ritornaNome() {
        return nome;
    }

    private String nome;
    private double saldo;

}

// ContoServer.java

import java.rmi.*;

public class ContoServer {

    public static void main(String[] args) {
        String nome = null;
        try {
            // creo ed esporto gli oggetti
            ContoImpl[] conto = new ContoImpl[args.length];
            for (int i = 0; i < args.length; i++)
                conto[i] = new ContoImpl(args[i]);

            // li registro presso il registry
            System.out.println(MSG_USR1);

            for (int i = 0; i < args.length; i++) {
                nome = conto[i].ritornaNome();
                Naming.bind("//localhost:1110/" + nome, conto[i]);
                System.out.println(nome);
            }
        }
        catch (RemoteException e) {
                System.err.println(MSG_ERR3);
                e.printStackTrace();
                System.exit(1);
        }
        catch (AlreadyBoundException e) {
                System.err.println(nome + MSG_ERR4);
        }
        catch(java.net.MalformedURLException e) {
                System.err.println(MSG_ERR5);
        }
    }
    private static final String MSG_USR1 = "Attesa delle invocazioni remote di metodi per gli oggetti: ";

    private static final String MSG_ERR3 = " Impossibile contattare il registry.";
    private static final String MSG_ERR4 = " e' un nome gia' associato ad un oggetto nel registry.";
    private static final String MSG_ERR5 = " URL errata. ";
}

// NegativeAmountException.java

public class NegativeAmountException extends Exception {
    public NegativeAmountException() {}
    public NegativeAmountException(String message) {
        super(message);
    }
}
```

Script di compilazione ed esecuzione (secondo la Fig.1):

```
rem make.bat
set CLASSPATH=
javac javaclient\*.java
set CLASSPATH=
javac javaserver\*.java
set CLASSPATH=
rmic -v1.2 -classpath javaserver -d javaserver ContoImpl
```

```
copy /Y javaserver\Conto.class .\rmiclasslocation
copy /Y javaserver\ContoImpl_Stub.class .\rmiclasslocation
copy /Y javaserver\NegativeAmountException.class .\rmiclasslocation
copy /Y javaclient\*.class .\rmiclient
copy /Y javaclient\policy.txt .\rmiclient
rem RMIregistry is empty
copy /Y javaserver\*.class .\rmiserver
pause
```

```
rem rmiregistry\go.bat
set CLASSPATH=
color 0F
rmiregistry 1110
pause
```

```
rem rmiserver\go.bat
set CLASSPATH=
color 1F
java -Djava.rmi.server.codebase=http://www2.ing.unipi.it/~o1553499/rmiclasslocation/ ContoServer mio tuo
rem con il web server Tomcat
rem java -Djava.rmi.server.codebase=http://10.114.233.15:8080/RMIclasslocation/ ContoServer mio tuo
pause
```

```
rem rmiclient\go.bat
set CLASSPATH=
color 2F
java -Djava.security.policy=policy.txt ContoClient localhost 100 50
rem server su host remoto
rem java -Djava.security.policy=policy.txt ContoClient 10.114.233.15 100 50
pause
```

Testing del programma:

• L'URL del codebase può essere anche un indirizzo di cartella locale, o condivisa in una LAN, es.:

`file://localhost/D:/TIGA/lab/rmiclasslocation/`

`file://localhost/A:/lab10/webserver/`

`file://169.254.60.70/lab/myclasses/`

`file://169.254.60.70/C:/lab/myclasses/`

oppure l'indirizzo di un server ftp (`ftp://`).

• Viene mostrato uno scenario di esecuzione standard. Ciascuna console viene aperta sull'host indicato fra parentesi.

Java Console (Server Host 131.121.191.162)

```
rmiregistry> set CLASSPATH=
rmiregistry> rmiregistry 1110
_
```

Java Console (Server Host 131.121.191.162)

```
set CLASSPATH=
java -Djava.rmi.server.codebase=http://www2.ing.unipi.it/~o1553499/rmiclasslocation/ ContoServer mio tuo
Attesa delle invocazioni remote di metodi per gli oggetti:
mio
tuo
rmiserver> _
```

Java Console (Client Host)

```
set CLASSPATH=
java -Djava.security.policy=policy.txt ContoClient 131.121.191.162 100 50

Elenco nomi registrati:
//131.121.191.162:1110/mio
//131.121.191.162:1110/tuo
Nuovo saldo di //131.121.191.162:1110/mio: 50.0

rmiclient> _
Elenco nomi registrati:
//131.121.191.162:1110/mio
//131.121.191.162:1110/tuo
```

```
Nuovo saldo di //131.121.191.162:1110/tuo: 50.0

rmiclient> _
Elenco nomi registrati:
//131.121.191.162:1110/mio
//131.121.191.162:1110/tuo
Nuovo saldo di //131.121.191.162:1110/mio: 100.0
```

ESERCITAZIONE TIGA: Architettura *Applet–RMI–Database*

Relativamente a quanto specificato nelle esercitazioni su *Swing*, *RMI*, e *JDBC*, si realizzi un servizio di conto corrente remoto strutturato secondo un'architettura a tre livelli *JavaApplet–RMI–Database* (Fig.4).

Il *front-end*[48] del sistema è costituito da una applet `BancaClient`, che realizza la *GUI* di Fig.1[49] ed implementa anche la funzione di client RMI invocando i metodi *accredita* (una quantità positiva o negativa) o *saldo* (che può anche essere negativo). Il *middleware*[50] è composto dalle classi `BancaServer`, il server RMI gestore degli oggetti `BancaImpl`, e `BancaImpl` medesima che implementa i servizi ricevendo richieste dei client ed accedendo al database mediante JDBC per soddisfarle. Infine, il *back-end*[51] è il DBMS *MySQL*© (a) oppure *Microsoft Access*© (b), che riceve le richieste da parte degli oggetti `BancaImpl` come statement *SQL* e restituisce i risultati, i quali riattraversano in senso inverso gli strati di software sino alla *GUI*.

Il database `banca` contiene i conti correnti in una tabella `CONTI`, dagli attributi `NOME` e `SALDO` (Fig.2). La classe `BancaImpl` (Fig.3) implementa una interfaccia locale (`BancaLocale`) oltre a quella remota (`Banca`).

Fig. 1 – *Graphical User Interface* del Cliente

CONTI

NOME	SALDO
pippo	100.50
pluto	3200.43
…	…

Fig. 2 - Tabella `CONTI` del database `banca`

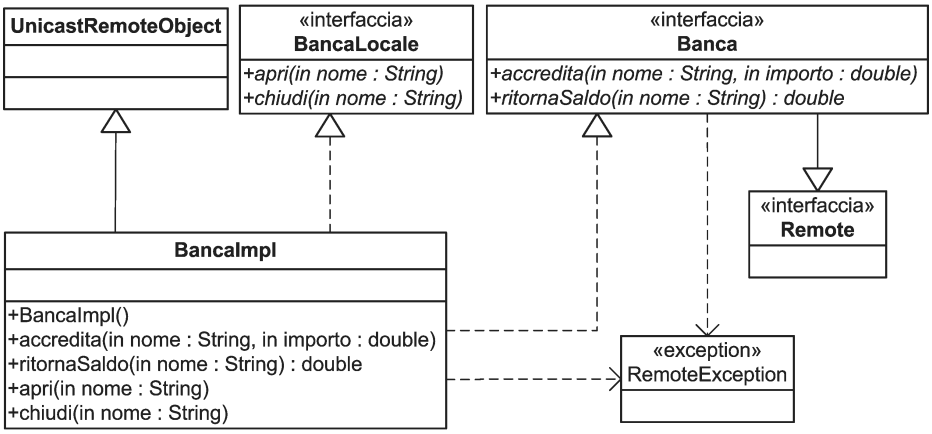


Fig. 3 – Diagramma delle Classi

```
public interface Banca extends Remote {
  void   accredita(String nome, double importo) throws RemoteException;
  double ritornaSaldo(String nome)              throws RemoteException;
}
```

```
public interface BancaLocale {
  void apri(String nome);
  void chiudi(String nome);
}
```

PROGETTARE E REALIZZARE:

Tutti i moduli descritti, semplificando al minimo la gestione delle eccezioni. Tralasciare l'implementazione dell'interfaccia utente locale creando staticamente due conti "pippo" e "pluto" all'interno del server RMI. Per semplicità il database si trova sul medesimo host di residenza di server e registry RMI (Fig.5) i quali devono necessariamente risiedere sul medesimo HTTP server da dove l'applet verrà scaricata[52]. Quindi il client accede ad un unico server, prima da browser mediante `http` per scaricare l'applet e le classi relative (comprese stub ed interfaccia) e poi attraverso `rmi` per comunicare con il registry ed il server.

[48] *Front-end*, letteralmente "frontale", "all'estremità più vicina a chi usa l'applicazione" e quindi "che fornisce un'interfaccia" o "che esegue funzioni di comunicazione con dispositivi esterni".
[49] È possibile avere importi e saldi negativi. Premendo *accredita* viene sommato l'importo inserito e restituito il saldo nella medesima casella di inserimento.
[50] *Middleware*, letteralmente "parte intermedia" , in generale é uno strato di software tra la rete e le applicazioni, che provvede a servizi come identificazione, autenticazione, autorizzazione, direttorio, sicurezza.
[51] *Back-end*, letteralmente "posteriore", "all'estremità più lontana da chi usa l'applicazione", o anche "riservato per compiti determinati o secondari che non occorre mostrare all'utente".
[52] Per motivi di sicurezza un'applet non può stabilire connessioni con altri host diversi da quelli da cui è stata scaricata, a meno che essa non venga "firmata" e quindi acquisisca maggiori privilegi sugli host che ne riconoscono la firma. Dunque nel presente esercizio si trascuri il caricamento dinamico di classi da un terzo http server.

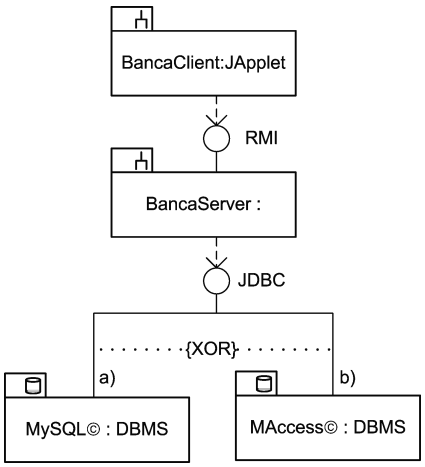


Fig. 4 – Sottosistemi funzionali

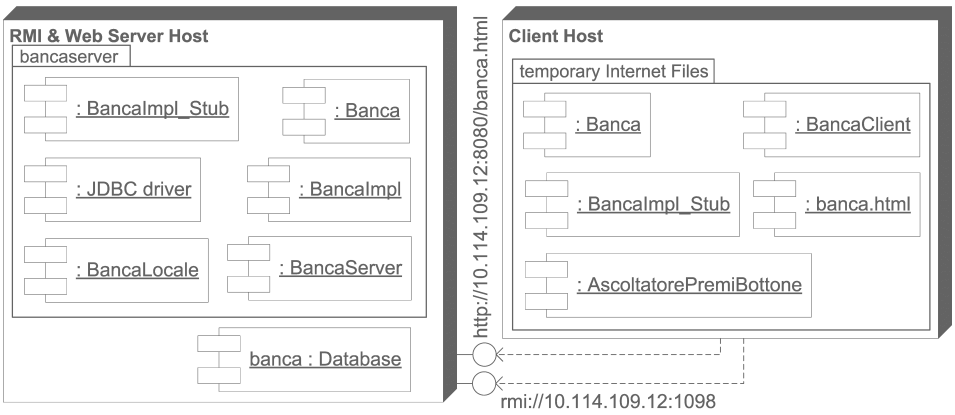


Fig. 5 – Diagramma di distribuzione dei componenti fisici

ESERCITAZIONE TIGA : Architettura *Applet–RMI–Database* – SOLUZIONE PROPOSTA

JDBC e Windows© :

In applicazioni Java/JDBC ogni connessione a un database viene gestita da un oggetto di classe `java.sql.Connection`, che viene restituito dal metodo di classe `java.sql.DriverManager.getConnection(URL)`, dove URL è la collocazione del database. Il DriverManager può restituire una connessione se è stato precedentemente caricato un driver JDBC adatto per quel database, mediante il metodo di classe `Class.forName(<nome del driver>)`. Ad esempio, le istruzioni:

```
Class.forName("org.gjt.mm.mysql.Driver");
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Class.forName("oracle.jdbc.driver.OracleDriver");
```

registrano i driver e caricano le classi rispettive (=contenitori dei driver) per *MySQL*®, *Microsoft Access*© ed *Oracle*©. Relativamente al primo DBMS, si avvia da riga di comando l'applicazione *c:\mysql\bin\mysql mysql -u root*, e poi si crea il database con lo statement *CREATE DATABASE banca;*. Infine, si lancia il server *c:\mysql\bin\winmysqladmin.exe* (oppure *mysqld.exe* per versioni precedenti di *Windows*). Relativamente al secondo DBMS, si crea un database vuoto *banca.mdb* [53], e poi si registra il database come *sorgente ODBC* [54] . Fatto ciò, è possibile gestire il database esclusivamente da Java, mediante statement DDL e DML[55], ad esempio:

```
Connection conn = DriverManager.getConnection(
       "jdbc:odbc:banca");                                    // Access
       "jdbc:mysql://localhost:3306/banca?user=root&password=");    // MySQL
 Statement  stat = conn.createStatement();
  stat.executeUpdate("INSERT INTO CONTI VALUES ('pippo', 0)");
 stat.close();
conn.close();
```

Le risorse impegnate da un oggetto `Statement` o `Connection` possono essere immediatamente liberate mediante una `close()`, oppure gestite automaticamente dal garbage collector. Quando viene chiuso un oggetto `Statement`, eventuali `ResultSet` associati sono automaticamente chiusi.

Problemi causati da installazioni multiple di JVM :

Spesso su un unico host risiedono diverse JVM: il pacchetto *J2SE SDK*, il pacchetto *J2SE JRE*, o *JRE* contenuto in pacchetti applicativi quali il DBMS *Oracle*©, oppure diverse versioni dei medesimi pacchetti. Ciascuna di queste installazioni inserirà nella variabile ambiente *PATH* (es. in cima o in fondo) il proprio percorso della cartella *bin*. Quando da console si digita semplicemente "java" o "javac" il sistema scorre la variabile ambiente *PATH* (da cima a fondo) alla ricerca dell'applicazione. Pertanto, digitando "javac" può essere eseguita l'applicazione della *J2SDK v. 1.0.5* (dal momento che le altre *JRE* non contengono "javac"), mentre digitando poi "java" può essere eseguita l'applicazione di *Oracle jre v. 1.1.0*, che per prima si trova nel *PATH*. Una JVM precedente può non essere in grado di eseguire codice compilato da una versione successiva di *SDK*, per cui può generarsi l'eccezione *"unsupported major.minor version ..."*. Altri problemi si hanno con RMI, in quanto la piattaforma java di Oracle abilita la *JAR caching* che impedisce a client e server RMI di comunicare direttamente con il registry. Per evitare ciò , si invochi sia "java" che "javac" indicando il full path[56].

Applet: parametri di ingresso e debugging:

I parametri del client applet non possono essere passati "da riga di comando", ma tramite la proprietà `<PARAM>` del tag `<APPLET>` presente nel codice `banca.html`. In particolare, il parametro *RMI_IP* dovrà contenere i) "*localhost*" per l'applet caricata localmente ii) l'indirizzo IP del server nel caso di applet caricata da web server, affinchè il Security Manager permetta tali connessioni.

Facendo partire una applet mediante `appletviewer banca.html`, si possono leggere sulla shell tutti i messaggi inviati a `System.out` o `System.err`, che generalmente non é opportuno visualizzare nello spazio della GUI, in modo da vedere ad esempio i messaggi prodotti da eventuali eccezioni durante la fase di sviluppo. La medesima console é disponibile da web browser. Installando[57] il *Java 2 Runtime Environment (J2RE)*, noto come *Java plug-in*, si avrà una sola JVM per i browser piú diffusi. Dopo l'installazione, avviando l'applicazione *Java Plug-in* da *pannello di controllo* di *Windows*© si possono configurare diverse caratteristiche, tra cui l'abilitazione della JVM per i singoli browser, oppure dalla scheda *Base* si può abilitare *Mostra console*. In tal caso, al caricamento dell'applet appare una finestra *console* per l'output.

Progettazione:

---

[53] Cliccando con il tasto destro del mouse, quindi selezionando nuovo, e poi *Applicazione di Microsoft Access*.

[54] Da *pannello di controllo*, aprire il pacchetto *Strumenti di Amministrazione*, quindi avviare l'applicazione *Origine Dati (ODBC)*, selezionare la scheda *DSN utente* o *DSN di sistema*, cliccare su *Aggiungi*, selezionare il *Microsoft Access Driver (*.mdb)*, cliccare su *Fine* e in conclusione assegnare "banca" come *nome origine dati*, da adoperare nel codice java, e cliccare su *Seleziona* per reperire il database nel file system.

[55] In questo laboratorio si suppone che anche la tabella sia preliminarmente creata. Entrambi i DBMS forniscono sia una intefaccia utente grafica (pacchetto *MySQL Query Browser* nel caso di *MySQL*) che una console per eseguire statement o script SQL (*queries→ new→ design view→ close→ SQL* nel caso di *MAccess*), per cui è possibile creare la tabella con il seguente statement: *CREATE TABLE banca.CONTI (NOME VARCHAR(64) NOT NULL, SALDO DOUBLE NOT NULL, PRIMARY KEY(NOME));*

[56] Es. C:\Programmi\Java\jdk1.5.0_01\bin\java …

[57] Da `http://java.sun.com/j2se/1.4.2/download.html`. Di default viene installato su C:\Programmi\Java\j2re1.4.2_03

In Fig.1 viene mostrata la distribuzione dei file. Nel caso di esecuzione dell'applet da web server, occorre posizionare la cartella *bancaclient* nella radice del server http[58].

Sostanzialmente `BancaServer` è un server RMI, con una struttura simile a `ContoServer` del laboratorio su *RMI*. Mentre `BancaClient` è una applet con la struttura di `MiaInterfaccia` del laboratorio su *Swing*, e qualche funzione di client RMI (`ContoClient` del laboratorio *RMI*), ossia nel metodo `init()` contiene una `lookup` per rintracciare l'oggetto remoto, ed in `eseguiOperazione()` l'invocazione dei metodi remoti come conseguenza dell'evento gestito (la pressione del bottone).

Quindi una prima parte di questo esercizio consiste nel *riuso di codice*. La novità consiste nell'accesso alla base di dati, gestito nei metodi di `BancaImpl` . Si noti che l'applet `BancaClient` è totalmente ignara della base di dati sottostante, in quanto comunica esclusivamente con il middleware, senza effettuare alcuna chiamata JDBC.

I metodi *accredita* e *ritornaSaldo* sono estremamente semplici: ciascuno richiede una sola operazione elementare (un'istruzione SQL) che può singolarmente fallire o riuscire, e in tal caso resa immediatamente permanente con la modalità auto-commit. Pertanto non è necessario adoperare transazioni, ossia non ha senso definire livelli di isolamento.

D'altra parte, l'esecuzione dei metodi relativi al medesimo oggetto `BancaImpl` gestore del database avviene sul server RMI. Un client potrebbe richiedere un servizio nel momento in cui il server RMI é ancora in fase di elaborazione per soddisfare una precedente richiesta. Le specifiche RMI non vietano la possibilità del multithreading, per cui è possibile che venga lanciato un thread che serva la nuova richiesta. Quindi, l'implementazione di un oggetto remoto deve sempre essere thread-safe. Nella fattispecie, sebbene i metodi remoti non siano dichiarati `synchronized`, la loro implementazione è thread-safe. Infatti la classe `BancaImpl` ha solo campi `final` (automaticamente thread-safe), e poi ciascun metodo istanzia localmente le proprie `Connection` i cui riferimenti faranno parte dello stack privato di ogni thread e riferiranno nello heap oggetti diversi. Quindi le diverse richieste di diversi host non verranno serializzate e potranno essere eseguite al massimo grado di parallelismo, a patto che il DBMS sia in grado di gestire connessioni multiple ed eseguire più statement in parallelo su record diversi della medesima tabella `CONTI` [59].

In generale, un'altra proprietà interessante delle connessioni multiple è che si possono gestire in parallelo transazioni indipendenti. Una transazione è una (breve) sequenza di operazioni sul DB con le proprietà *ACID*[60]. Poichè spesso transazioni concorrenti operano sugli stessi dati, si ripresentano le anomalie tipiche della programmazione concorrente, per cui occorrono degli strumenti meno problematici dei "lock" per sincronizzare gli accessi in diverse modalità. Le *API JDBC* permettono di configurare i *livelli di isolamento*[61] delegando al DBMS la gestione dei lock ed il rilevamento e recupero da situazioni di stallo.

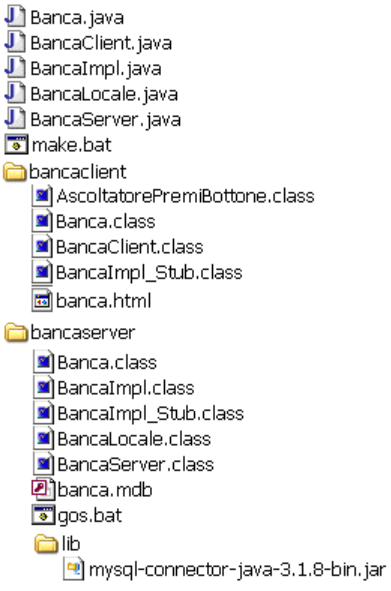


Fig.1 – Distribuzione dei file **.**

Codifica in linguaggio Java:

```
<!--banca.html-->
<HTML>
  <HEAD>
    <TITLE> BANCA ON LINE </TITLE>
  </HEAD>
  <BODY bgcolor = "blue" text = "white">
    <DIV align = "center" >
      <H2>ESERCITAZIONE TIGA: Architettura Applet-RMI-Database</H2>
      <H3>Sportello bancario per i clienti "pippo" e "pluto"</H3>
      <APPLET code = "BancaClient.class" width = "300" height = "100">
        <PARAM name = "RMI_IP"    value = "localhost">
        <PARAM name = "RMI_PORT"  value = "1098">
        <PARAM name = "BANK_NAME" value = "bancaOnLine">
      </APPLET>
    </DIV>
  </BODY>
</HTML>
```

---

[58] Ad esempio, adoperando il server web *Internet Information Services© (IIS)* tale cartella è `C:\Inetpub\wwwroot\`, mentre adoperando *Apache Tomcat®* è `C:\Program Files\Apache Software Foundation\Tomcat 5.0\webapps\ROOT\`.

[59] Provare il seguente esperimento: porre il database *MAccess* su un floppy e registrarlo come sorgente dati ODBC, quindi accedere in contemporenea con numerosi client. Il DBMS in tal caso sarà costretto ad attendere i lunghi tempi di risposta del floppy. Per cui sarà il disco a fungere da "collo di bottiglia", mentre la percentuale di utilizzo della CPU (visualizzabile mediante il Task Maneger) da parte della JVM sarà bassa (5%). Tempi di risposta simili possono esserci in un database reale, con query su tabelle di migliaia di tuple, a prescindere dalle gerarchie di memoria e dalla potenza della CPU. Questo schema viene detto "molto I/O poco calcolo", in cui la piattaforma Java trova impiego ottimale dal momento che lo scopo consiste nell'accedere a dati piuttosto che elaborarli con algoritmi computazionalmente pesanti.

[60] **Atomicità**: o tutte le operazioni terminano con successo (commit) oppure, se anche una sola di esse fallisce, l'intera transazione viene riavvolta (rollback). **Consistenza**: una transazione realizza un passaggio del DB tra due stati consistenti. **Isolamento**: l'effetto di esecuzioni concorrenti di più transazioni deve essere equivalente ad una esecuzione seriale delle stesse. Quindi, transazioni concorrenti non devono influenzarsi reciprocamente. **Durabilità**: gli effetti sulla base di dati prodotti da una transazione terminata con successo sono permanenti (committed) , cioè non sono compromessi da eventuali malfunzionamenti.

[61] Introdotto in *ANSI SQL-92*. Il più alto livello di isolamento e la minima efficienza consistono nella serializzazione di tutte le transazioni.

```
// Banca.java
import java.rmi.*;
public interface Banca extends Remote {
    void        accredita (String nome, double importo)  throws RemoteException;
    double      ritornaSaldo (String nome)               throws RemoteException;
  }

// BancaLocale.java
public interface BancaLocale {
    void apri (String nome);
    void chiudi (String nome);
}

// BancaImpl.java
import java.sql.*;
import java.rmi.*;
import java.rmi.server.*;

public class BancaImpl  extends UnicastRemoteObject implements Banca, BancaLocale {

    public BancaImpl() throws RemoteException {
        try {
            Class.forName(dbDriver);
        }
        catch (ClassNotFoundException e) {}
    }
    public void apri(String nome) {
        try {
           Connection conn = DriverManager.getConnection(dbUrl);
           PreparedStatement stat = conn.prepareStatement(createConto);
             stat.setString(1, nome);
             stat.executeUpdate();
            stat.close();
           conn.close();
        }
        catch (SQLException e) { gestoreEccezioneSQL(e); }
    }
    public void chiudi(String nome) {
        try {
           Connection conn = DriverManager.getConnection(dbUrl);
            PreparedStatement stat = conn.prepareStatement(deleteConto);
             stat.setString(1, nome);
             stat.executeUpdate();
            stat.close();
           conn.close();
        }
        catch (SQLException e) { gestoreEccezioneSQL(e); }
    }
    public double ritornaSaldo(String nome) throws RemoteException {
        double result = 0.0;
        try {
           Connection conn = DriverManager.getConnection(dbUrl);
            PreparedStatement stat = conn.prepareStatement(readSaldo);
             stat.setString(1, nome);
              ResultSet rset = stat.executeQuery();
              if (rset.next())
                result = rset.getDouble("SALDO");
             rset.close();
            stat.close();
           conn.close();
        }
        catch (SQLException e) { gestoreEccezioneSQL(e); }
        return result;
    }
    public void  accredita (String nome, double importo)  throws RemoteException {
        try {
          Connection conn = DriverManager.getConnection(dbUrl);
            PreparedStatement stat = conn.prepareStatement(updateSaldo);
             stat.setDouble(1, importo);
             stat.setString(2, nome);
             stat.executeUpdate();
            stat.close();
           conn.close();
        }
        catch (SQLException e) { gestoreEccezioneSQL(e); }
    }

    private static void gestoreEccezioneSQL(SQLException e) {
        System.err.println("Message: " + e.getMessage() );
        System.err.println("SQLState: " + e.getSQLState() );
        System.err.println("ErrorCode: " + e.getErrorCode() );
        e.printStackTrace();
    }

    private static final String  dbDriver =
                 //"sun.jdbc.odbc.JdbcOdbcDriver";                          // Microsoft Access
                 "org.gjt.mm.mysql.Driver";                                 // MySQL
```

```
    private static final String     dbUrl =
                //"jdbc:odbc:banca";                                        // Microsoft Access
                "jdbc:mysql://localhost:3306/banca?user=root&password=";    // MySQL

    private static final String createConto =
        "INSERT INTO CONTI VALUES (?, 0)";
    private static final String deleteConto =
        "DELETE FROM CONTI WHERE NOME = ?";
    private static final String readSaldo =
        "SELECT SALDO FROM CONTI WHERE NOME = ?";
    private static final String updateSaldo =
        "UPDATE CONTI SET SALDO = SALDO + ? WHERE NOME = ?";
}
// BancaServer.java
import java.rmi.*;
import java.rmi.registry.*;
public class BancaServer {

    public static void main(String[] args) {
        String nome = args[0];
        try {
            LocateRegistry.createRegistry(1098);

            // creo ed esporto la banca
            BancaImpl   banca1 = new BancaImpl();
            Banca       bancaOnLine  = banca1;
            BancaLocale bancaOffLine = banca1;

            bancaOffLine.apri("pippo");
            bancaOffLine.apri("pluto");

            // la registro presso il registry
            Naming.bind("//localhost:1098/" + nome, bancaOnLine);

            System.out.println(MSG_USR1);
        }
        catch (RemoteException e) {
                System.err.println(MSG_ERR3);
                e.printStackTrace();
                System.exit(1);
        }
        catch (AlreadyBoundException e) {
                System.err.println(nome + MSG_ERR4);
        }
        catch(java.net.MalformedURLException e) {
                System.err.println(MSG_ERR5);
        }
    }
    private static final String MSG_USR1 = "Attesa delle invocazioni remote... ";
    private static final String MSG_ERR3 = " Impossibile contattare il registry.";
    private static final String MSG_ERR4 = " e' un nome gia' associato ad un oggetto nel registry.";
    private static final String MSG_ERR5 = " URL errata. ";
}
// BancaClient.java
import java.rmi.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class BancaClient extends JApplet {
    public void init() {

        url    = "//" + getParameter("RMI_IP") + ":" + getParameter("RMI_PORT")+ "/";
        nome   = getParameter("BANK_NAME");
        inizializzaGUI();
        try {
            bancaOnLine = (Banca)Naming.lookup(url + nome);
        }
        catch (NotBoundException e) {
            System.err.println(nome + MSG_ERR1);
            e.printStackTrace();
        }
        catch (RemoteException e) {
            System.err.println(MSG_ERR2);
            e.printStackTrace();
        }
        catch(java.net.MalformedURLException e) {
            System.err.println(BancaServer.MSG_ERR5);
            e.printStackTrace();
        }
    }
    private void inizializzaGUI() {
        JPanel pannello = new JPanel();

        pannello.setLayout( new GridLayout(3,2) );

        campoNome = new JTextField("");
        campoImpo = new JTextField("0.00");
        bottoOper = new JButton(" ACCREDITA");
```

```
        // gestore di eventi
        ActionListener gestore = new AscoltatorePremiBottone(this);

        // associa componenti a gestore
        bottoOper.addActionListener(gestore);

        // attacca componenti al pannello
        pannello.add(etich[0]);
        pannello.add(campoNome);
        pannello.add(etich[1]);
        pannello.add(campoImpo);
        pannello.add(etich[2]);
        pannello.add(bottoOper);

        // attacca pannello alla Frame
        getContentPane().add(pannello);
    }
        public void eseguiOperazione() {

        String nome     = campoNome.getText();
        double importo  = Double.parseDouble(campoImpo.getText());

        try {
            if (importo!=0)
                bancaOnLine.accredita(nome,importo);
            campoImpo.setText(Double.toString(bancaOnLine.ritornaSaldo(nome)));
        }
        catch (java.security.AccessControlException e) {
            System.err.println(MSG_ERR3);
            e.printStackTrace();
        }
        catch (RemoteException e) {
            System.err.println(MSG_ERR2);
            e.printStackTrace();
        }
    }
        // campi interfaccia
    private JTextField  campoNome;
    private JTextField  campoImpo;
    private JButton     bottoOper;

    private static final JLabel[] etich = {
        new JLabel("nome ", JLabel.RIGHT),
        new JLabel("importo ", JLabel.RIGHT),
        new JLabel("operazione ", JLabel.RIGHT)
    };

    // campi logici
    private Banca       bancaOnLine;

    private String url ;
    private String nome ;

    private static final String MSG_ERR1 = " e' un nome non associato ad alcun oggetto nel registry";
    private static final String MSG_ERR2 = " Impossibile contattare il registry.";
    private static final String MSG_ERR3 = " Operazione non permessa.";
}

class AscoltatorePremiBottone implements ActionListener {
    public AscoltatorePremiBottone(BancaClient interfaccia) {
        interf = interfaccia;
    }
    public void actionPerformed(ActionEvent e) {
        interf.eseguiOperazione();
    }
    private BancaClient interf;

}
```

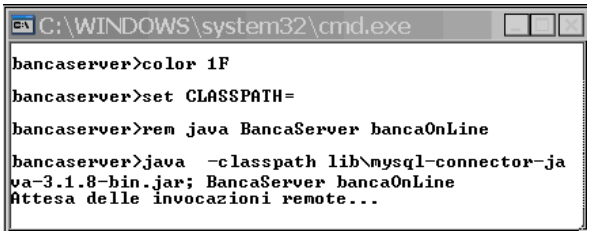


Fig.2 – Esecuzione standard del server.

```
-----------------------------------------------------------------------
rem make.bat
set CLASSPATH=
rem MSAccess
rem javac *.java
javac -classpath lib\mysql-connector-java-3.1.8-bin.jar; *.java
rmic -v1.2 BancaImpl

copy /Y Banca.class .\bancaclient
copy /Y BancaImpl_Stub.class .\bancaclient
copy /Y AscoltatorePremiBottone.class .\bancaclient
copy /Y BancaClient.class .\bancaclient

copy /Y *.class .\bancaserver
del  bancaserver\BancaClient.class
del  bancaserver\AscoltatorePremiBottone.class

pause
-----------------------------------------------------------------------
rem gos.bat
```

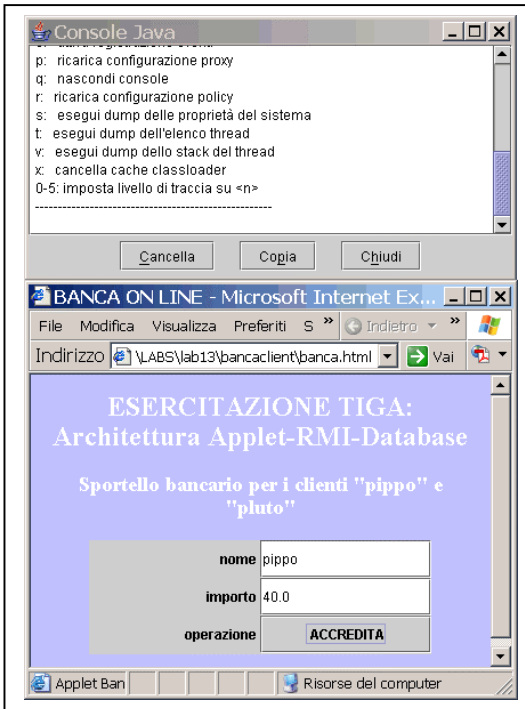


Fig.3 – Esecuzione standard dell'applet.

```
color 1F
set CLASSPATH=
rem MSAccess
rem java BancaServer bancaOnLine
java  -classpath lib\mysql-connector-java-3.1.8-bin.jar; BancaServer bancaOnLine
-----------------------------------------------------------------
```

Testing del programma:

- In Fig.2-3 viene mostrato uno scenario di esecuzione standard.