Esercitazioni di Laboratorio

9. Laboratorio 9

Esercizio 0

E' attivo il quiz di Autovalutazione n.4.
E' ancora attivo anche il questionario per il feedback sul corso!

L'utilizzo di scanner puo' essere un po' ostico. Se avete difficolta' potete studiare il seguente esercizio svolto. Chi si sentisse sicuro abbastanza puo' ovviamente svolgerlo autonomamente e poi verificare la propria soluzione o passare agli esercizi successivi.

Esercizio 1 - svolto - ripasso utilizzo di Scanner

Argomento: lettura da file, utilizzo di Scanner per lettura da file e tokenizzazione di stringhe

Scrivere una classe eseguibile WordCount che conti righe e parole di un file di testo.
Per prima cosa scrivere la struttura della classe (essendo eseguibile dovro' metterci il metodo main). Salvo tutto so WordCount.java

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

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

   }
}
```

Il programma acquisisce il nome del file da linea di comando, attraverso l'array args.
Verificare che ci sia un elemento da leggere e assegnarlo ad una stringa filename. Dichiarare e inizializzare due variabili contatore, una per le righe e una per le parole.

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

public class WordCount{
   public static void main(String[] args){
      //leggo il nome del file da linea di comando
      if(args.length != 1){
         System.out.println("Usage: javac WordCount nomefile");
         System.exit(1);
      }
      // dichiaro e inizializzo le variabili
      String filename = args[0];
      int wordCount = 0;
      int lineCount = 0;

   }
}
```

Ora devo aprire il file e uno scanner per poterlo leggere. Utilizzo try-with-resources, che abbiamo visto a lezione essere piu' pratico ed evitare problemi con il close(). Metto anche il messaggio di stampa finale.

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

public class WordCount{
   public static void main(String[] args){
      //leggo il nome del file da linea di comando
      if(args.length != 1){
         System.out.println("Usage: javac WordCount nomefile");
         System.exit(1);
      }
      // dichiaro e inizializzo le variabili
      String filename = args[0];
      int wordCount = 0;
      int lineCount = 0;

      try (FileReader reader = new FileReader(filename);
                Scanner scan = new Scanner(reader)){

         // qui mettero' la lettura


      }
      catch(Exception e){
         System.out.println(e);
      }
      System.out.println("Il file "+filename+" contiene "+
                     lineCount+" righe e "+wordCount+" parole");
   }
}
```

Ora leggo una riga alla volta e incremento il contatore corrispondente. Per fare questo devo utilizzare un ciclo while nel quale controllo che ci sia una prossima riga da leggere attraverso hasNextLine(). Questo metodo controlla solo se c'e' la riga (restituisce true/false), non la "estrae" dal flusso di scansione. Per fare questo devo usare nextLine() che mi restituisce una stringa!

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

public class WordCount{
   public static void main(String[] args){
      //leggo il nome del file da linea di comando
      if(args.length != 1){
         System.out.println("Usage: javac WordCount nomefile");
         System.exit(1);
      }
      // dichiaro e inizializzo le variabili
      String filename = args[0];
      int wordCount = 0;
      int lineCount = 0;

      try (FileReader reader = new FileReader(filename);
                Scanner scan = new Scanner(reader)){

          while(scan.hasNext()){
            // se sono nel ciclo vuol dire che c'e'
            // una riga disponibile e posso leggerla

            // Leggo la riga intera con nextLine() e incremento
            // il contatore di righe.
            String line = scan.nextLine();
            lineCount++;

          }
      }
      catch(Exception e){
         System.out.println(e);
      }
      System.out.println("Il file "+filename+" contiene "+
                     lineCount+" righe e "+wordCount+" parole");
   }
}
```

Compilate ed eseguite dando come nome del file di input proprio quello che state scrivendo!
javac WordCount.java
java WordCount WordCount.java

Il vostro programma contera' le righe del suo codice sorgente. Verificate che il numero in output corrisponda al numero di righe che appare nel vostro editor.

Ora aggiungiamo un ulteriore ciclo while, interno al precedente, che prende ogni riga e la scansiona parola per parola. Per fare questo avro' bisogno di un altro scanner che sara' inizializzato con la stringa line. Per verificare se ci sono altre parole ora devo usare hasNext(), che restituisce true/false, come condizione di un ciclo all'interno del quale leggo ciascuna parola con next() e incremento il contatore di parole.

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

public class WordCount{
   public static void main(String[] args){
      //leggo il nome del file da linea di comando
      if(args.length != 1){
         System.out.println("Usage: javac WordCount nomefile");
         System.exit(1);
      }
      // dichiaro e inizializzo le variabili
      String filename = args[0];
      int wordCount = 0;
      int lineCount = 0;

      try (FileReader reader = new FileReader(filename);
                Scanner scan = new Scanner(reader)){

          while(scan.hasNext()){
            // se sono nel ciclo vuol dire che c'e'
            // una riga disponibile e posso leggerla

            // Leggo la riga intera con nextLine() e incremento
            // il contatore di righe
            String line = scan.nextLine();
            lineCount++;

            // La riga e' composta da piu' parole. Uso uno scanner
            // sulla riga per leggere una parola alla volta
            try(Scanner lineScanner = new Scanner(line)){
               // ora uso hasNext() per vedere se c'e' una
               // prossima parola da leggere
               while(lineScanner.hasNext()){
                  //se sono qui c'e' la parola da leggere
                  //la leggo con next() e incremento il contaparole
                  //NB: se non devo elaborare la parola, ma solo
                  //contarla, non serve assegnare il risultato di
                  //lineScanner.next() ad una variabile.
                  lineScanner.next();
                  wordCount++;
               }
            }
            catch(NoSuchElementException e){
               System.out.println("Errore in lettura delle parole");
            }

          }
      }
      catch(Exception e){
         System.out.println(e);
      }
      System.out.println("Il file "+filename+" contiene "+
                     lineCount+" righe e "+wordCount+" parole");
   }
}
```

Compilate ed eseguite. Ora anche il contatore di parole dovrebbe dare un risultato diverso da zero. Potete provare anche con altri file in input, piu' brevi, per verificare che il numero di parole sia corretto.

Esercizio 2 - ripasso in/out

Argomento: lettura e scrittura su file

Scrivere un programma che legga da standard input il nome di un file di input e di un file di output. Predisporre il file di input in modo che contenga parole (o brevi frasi) su diverse righe. Leggere il contenuto del file di input e scrivere sul file di output ciascuna riga rovesciata (prevedere un metodo statico per l'inversione della stringa corrispondente alla riga). Gestire tutte le eccezioni obbligatorie. Provare a verificare

cosa succede se si da in ingresso come nome del file di input un file inesistente.

Esercizio 3 - ripasso progettazione classi

Argomento: progettazione e collaudo di classi

Scrivere un programma per la risoluzione di equazioni di secondo grado $\mathbf{ax^2 + bx + c = 0}$. La realizzazione del programma va articolata in due fasi:

1. Progettare e scrivere una classe **QuadraticEquation**, il cui "scheletro" si trova a questo link. Questa interfaccia lascia aperte alcune scelte di progetto, che ciascuno potrà effettuare in autonomia, eventualmente aggiungendo altri metodi e/o campi di esemplare alla classe. In particolare:
    - Si puo` decidere che un oggetto di tipo **QuadraticEquation** sia *immutabile*, ovvero non possa piu` essere modificato una volta creato. Oppure si può dare la possibilità (tramite scrittura di opportuni *metodi modificatori*) di modificare oggetti di tipo **QuadraticEquation** già esistenti
    - Si puo` prevedere la presenza di *metodi di accesso*, che restituiscano informazioni sullo stato di un oggetto di tipo **QuadraticEquation**
    - Si puo` migliorare il progetto in modo da gestire correttamente i seguenti casi degeneri
        - **a=b=0** e **c!=0** (l'equazione non ha soluzioni)
        - **a=0**, **b!=0** (l'equazione ha una soluzione)
        - **a=b=c=0** (l'equazione ha infinite soluzioni)
2. Scrivere un programma di collaudo **QuadraticEquationTester** che utilizzi la classe **QuadraticEquation**. Il programma deve
    - leggere in ingresso i valori dei parametri a, b, c
    - (se esistono soluzioni reali) visualizzare le due soluzioni reali
    - (se non esistono soluzioni reali) visualizzare un messaggio di segnalazione

Esercizio 4

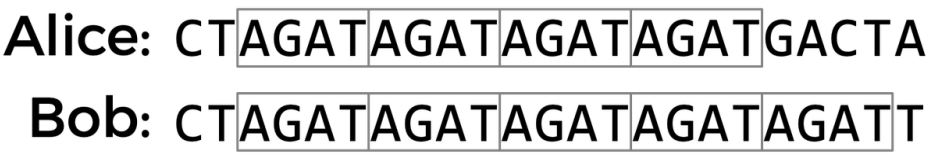
Siete stati incaricati dal Reparto Investigazioni Scientifiche di analizzare il DNA ritrovato in 3 casi insoluti e confrontarlo con quello di 3 indagati per scoprire il colpevole. Ad oggi avete acquisito tutte le conoscenze necessarie per poter implementare un programma di questo tipo: sapete realizzare le classi, sovrascrivere i metodi di Object, rendere eseguibile una classe, leggere da file, analizzare le stringhe, utilizzare i cicli, anche annidati, e prendere delle decisioni con gli if.

Il progetto e' un po' impegnativo, ma dovete anche cominciare ad abituarvi a scrivere codice che mette insieme diverse cose che abbiamo visto. Tuttavia, per chi avesse bisogno di un punto o da cui partire o di una guida da seguire, ho aggiunto un file alla file della consegna con delle indicazioni. Chi ritiene di non aver bisogno di guida puo' semplicemente ignorare il file.

Background

Il DNA, portatore dell'informazione genetica negli esseri viventi, e' utilizzato nelle investigazioni scientifiche da decenni. Ma come, esattamente, funziona la profilazione del DNA? Dato un campione di DNA, come riescono gli investigatori forensi a identificare a chi appartiene?

Il DNA e' in realta' composto da una sequenza di molecole, dette nucleotidi, che formano una doppia elica. Ogni nucleotide di DNA contiene una tra quattro basi: adenine (A), cytosine (C), guanine (G), or thymine (T). Ogni cella umana contiene miliardi di questi nucleotidi sistemati in sequenza. Alcune porzioni di questa sequenza, che altro non e' che il genoma, sono le stesse o sono molto simili, a tutti gli esseri umani. Tuttavia, altre porzioni di sequenza hanno una diversita' genetica piu' alta e quindi variano tra individui di una stessa specie. Uno dei posti dove il DNA tende ad avere un'elevata diversita' genetica e' nelle Short Tandem Repeats (STRs). Una STR e' una breve sequenza di DNA che tende a ripetersi piu' volte consecutivamente. Il numero di volte in cui una particolare STR si ripete varia molto tra individui diversi. Per esempio nell'immagine qui sotto Alice ha la STR "AGAT" ripetuta 4 volte nel suo DNA, mentre Bob ha la stessa STR ripetuta 5 volte.



La probabilita' che due individui diversi abbiano lo stesso numero di ripetizioni per una stessa STR e' molto basso. Se poi si usano piu' STR, invece di una, possiamo migliorare l'accuratezza della profilazione del DNA, riducendo ancora di piu' questa probabilita'. I database dell'FBI,

per esempio, usano 20 diversi STR, con una probabilita' che due DNA coincidano "per caso" pari o inferiore a uno su 1000000000000000000 (ovvero 10^-18).

Semplificando un po' le cose possiamo immaginare questi database come dei file, dove ogni riga corrisponde ad un individuo e ne riporta il nome, seguito dal numero di STR rilevate per ciascuna STR considerata e descritta nella prima riga.

```
name    AGAT AATG TATC
Alice    28    42   14
Bob      17    22   19
Charlie 36    18   25
```

Nell'esempio sopra, Alice ha la sequenza AGAT ripetuta 28 volte consecutive da qualche parte nel suo DNA, la sequenza AATG ripetuta 42 volte e TATC ripetuta 14 volte. Bob ha queste stesse sequenze ripetute 17, 22 e 19 volte, rispettivamente. Charlie invece 36, 18 e 25 volte.

Quindi, data una sequenza di DNA, che per noi altro non e' che una stringa, possiamo individuare a chi appartiene utilizzando tecniche bioinformatiche, ovvero sviluppando un algoritmo che analizzi tale sequenza e le informazioni presenti nel "database". In particolare si dovra':

- Analizzare la sequenza di caratteri che rappresenta il DNA ritrovato nella scena del crimine cercando il numero di occorrenze consecutive di ciascuna STR presente nel database investigativo;
- Confrontare i valori trovati (= profilo) con quelli corrispondenti agli individui "schedati" e se c'e' corrispondenza allora c'e' una buona probabilita' che quell'individuo sia il colpevole.

In realta' l'analisi e' un po' piu' complessa di come descritto qui, ma accontentiamoci!

Specifiche

Scrivere un programma eseguibile DNAProfile.java che identifica a chi appartiene la sequenza di DNA.

Il programma richiedera' due argomenti da riga di comando: il nome del file che contiene i profili degli indagati e il nome del file che contiene la sequenza di DNA da identificare.

La prima riga del file dei profili conterra' la parola "name" e poi le sequenze delle STR considerate. Le righe successive conterranno il nome dell'individuo seguito dalla numerosita' delle STR rilevate nel suo DNA.

Per ciascuna sequenza STR il programma dovra' calcolare la piu' lunga run di ripetizioni consecutive della STR nel DNA da identificare. Questa e' la parte, dal punto di vista algoritmico piu' complessa e sulla quale vi invito a ragionare su carta (potete utilizzare comunque i metodi della classe String, ad esempio substring).

Se i conteggi di STR hanno un match con qualche profilo noto, il nome del presunto colpevole dovra' essere riportato in uscita, altrimenti si scrivera' "nessun match trovato". Potete assumere che il conteggio di STR avra' un match con al massimo un individuo.

Utilizzando i file linkati sotto, se pensate possa semplificare, potete assumere inizialmente che il numero di individui sia 3 e che il numero di STR sia 3, poi potete generalizzare in un secondo momento.

Utilizzo:

java DNAprofile fileProfile fileDNA

File:

sospetti.txt
caso1.txt
caso2.txt
caso3.txt
caso4.txt

Soluzione guidata

credits: ispirato da nifty assignments

Esercizio 5

Argomento: ereditarietà, sovrascrivere i metodi della classe Object

Si consideri la gerarchia di ereditarietà costituita dalla superclasse **BankAccount** e dalle sue due sottoclassi **SavingsAccount** e **CheckingAccount** (i file completi saranno disponibili in "File Utili e soluzioni degli esercizi (Laboratorio 9)").

BankAccount e SavingsAccount sono state viste in classe, CheckingAccount e' una classe derivata da BankAccount in cui si tiene conto del numero di transazioni. Se si supera un certo limite si paga un costo per ogni transazione. Studiare il codice della classe.

Si completi la scrittura di queste tre classi sovrascrivendo in ciascuna di esse i metodi **toString** e **equals**, ereditati dalla superclasse universale **Object**. Per il metodo **toString** in particolare si richiede di adottare la convenzione della libreria standard.
Si verifichi il corretto funzionamento di tutti i metodi delle tre classi usando la classe di collaudo **AccountTester**. In particolare, i metodi **toString** sono corretti se le stampe a standard output dei vari oggetti rispettano le convenzioni della libreria standard; i metodi **equals** sono corretti se i controlli di uguaglianza forniscono i risultati attesi. La classe di collaudo utilizza un costrutto che non abbiamo visto ma il cui nome e' piuttosto intuitivo: a.getClass().getName() restituisce il nome della classe a cui appartiene l'oggetto a cui la variabile "a" si riferisce.

Esercizio 6

Argomento: ereditarietà, sovrascrivere metodi, metodi polimorfici

Si consideri la gerarchia di ereditarietà costituita dalla superclasse **BankAccount** e dalle sue due sottoclassi **SavingsAccount** e **CheckingAccount** (i file completi saranno disponibili in "File Utili e soluzioni degli esercizi (Laboratorio 9)").

Si richiede di aggiungere a questa gerarchia di ereditarieta` una nuova classe **TimeDepositAccount** ("conto di deposito vincolato"). Un conto di deposito vincolato e` identico a un conto di risparmio, a parte il fatto che l'intestatario si impegna a lasciare il denaro sul conto per un certo numero di mesi (definito all'apertura del conto insieme al tasso di interesse), senza fare prelievi durante questo periodo iniziale. E` prevista una penale di 20€ in caso di prelievo anticipato.
Completare la classe **TimeDepositAccount** sovrascrivendo in essa i metodi **toString** e **equals**, ereditati dalla sua superclasse. Effettuare il collaudo utilizzando la classe **AccountTester2**, dopo averla opportunamente studiata.

Suggerimenti *(da non leggere subito)*:

- La definizione di **TimeDepositAccount** appena data ci fa capire come questa nuova classe si deve inserire nella gerarchia di ereditarietà. Sarà una sottoclasse di ...
- Scrivere almeno due costruttori di **TimeDepositAccount**: uno che consente di inizializzare il tasso di interesse e i mesi di deposito vincolato, e uno che oltre a questi due parametri consente di inizializzare anche il saldo ad un valore non nullo.
- Il metodo **addInterest** viene invocato alla fine di ogni mese per accreditare gli interessi. Quindi il numero di mesi di deposito vincolato si deve ridurre ad ogni invocazione di questo metodo.
- Se al momento di un prelievo il numero di mesi di deposito vincolato e' ancora positivo, allora bisogna addebitare la penale