**Esercizi**

**1**.

**Soluzione**: Si veda il codice del file sorgente Gettysburg.java

**2**.

**Soluzione**: Si veda il codice del file sorgente Gettysburg2.java

**4**.

**Soluzione**: Si veda il codice del file sorgente RegistraVendita.java

**5**.

**Riferimenti:** Esercizio 13 del Capitolo 13

**Soluzione**: Si veda il codice del file sorgente CronometroGiri.java

**6**.

Si usi il file numeri.txt

**Soluzione**: Si veda il codice dei file sorgente NumeroDiTelefono.java, NumeroTelefonicoNonValidoException.java e NumeroDiTelefonoDemo.java

**7**.

Si usi il file contatti.txt

**Soluzione**: Si veda il codice dei file sorgente InformazioniContatto.java, InformazioniContattoDemo.java e ContattoNonValidoException.java

**8**.

Si usi il file parole.txt

**Soluzione**: Si veda il codice del file sorgente RimozionePrimaParola.java

**10**.

**Soluzione**: Si veda il codice del file sorgente CopiaFile.java

**12**.

Il codice in UTFBinaryFileWriter permette di creare un file con il formato appropriato.

**Soluzione**: Si veda il codice dei file sorgente UTFBinaryFileWriter.java e UTFBinaryFileReader

**13**.

**Soluzione**: Si veda il codice del file sorgente SalvaSegnale.java

**14**.

**Soluzione**: Si veda il codice del file sorgente RecuperaSegnale.java

**15**.

**Soluzione**: Si veda il codice del file sorgente ByteReader.java

\*\*\*\*\*\*

**Programmi**

**1.**

Questo progetto è ingannevole: richiede più programmi di quelli elencati nella definizione del problema per elaborare file binari e file di testo contenenti numeri in virgola mobile. Sono inoltre necessari programmi aggiuntivi per creare e visualizzare i file di dati.

La soluzione qui mostrata ha programmi separati per creare e visualizzare i file di dati, uno per il testo e l'altro per i file binari. Il programma per elaborare i file di testo richiede un metodo per tradurre i numeri da una stringa a double. Il modo più semplice è quello di utilizzare il metodo parseDouble della classe wrapper Double.

1) Sviluppare la classe per creare e visualizzare i file di dati binari. Buoni modelli da seguire sono Raddoppia, Listato 14.9, per l'organizzazione generale del programma, e FileBinarioOutputDemo, Listato 14.6, per mostrare il contenuto del file.

2) Sviluppare la classe per elaborare i dati in un file binario (visualizzare il più alto, il più basso e la media), sempre utilizzando Raddoppia, Listato 14.9, come modello.

Utilizzare FileDiTestoOutputDemo, Listato 14.1, e FileDiTestoInputConScannerDemo, Listato 10.2, come modelli per scrivere e leggere da file di testo.

**Riferimenti:** Listato 14.1, Listato 14.2, Listato 14.6, Listato 14.9

**Soluzione**: Si veda il codice dei file sorgente NumeriRealiAltoBassoMediaBinario.java, NumeriRealiAltoBassoMediaTesto.java, ScriviNumeriRealiFileDiTesto.java e ScriviNumeriRealiFileBinario.java

**2**.

Questo progetto è simile al Programma 1: i file di dati devono essere creati prima di poter effettuare qualsiasi elaborazione ed è più facile lavorare con i file binari che con il testo. Quindi un buon approccio è iniziare con le classi che trattano i file binari del Progetto 1 e modificarle in modo che i file di dati possano essere creati e visualizzati, quindi scrivere il programma per elaborare i file di dati binari. Si noti che è necessario definire un programma separato per visualizzare i file di dati e i file creati dal programma che rimuove i duplicati. Se non è stato svolto il Progetto, allora è possibile iniziare con i file di testo dei Listati 14.6 e 14.9. Dopo che tutti i programmi di file binari sono stati scritti e testati, è più facile sviluppare i programmi per creare, leggere ed elaborare i file di testo. Proprio come per il Progetto 1, il codice per impostare i flussi di input e output deve essere cambiato, la condizione del ciclo while deve essere cambiata per terminare quando viene letta una stringa nulla, e le stringhe di testo nel file di dati devono essere cambiate in int usando il metodo parseInt nella classe Integer.

**Riferimenti:** Programma 1, capitolo 14, Listato 14.6, Listato 14.9

**Soluzione**: Si veda il codice dei file sorgente ScriviNumeriInteriFileDiTesto.java, ScriviNumeriInteriFileBinario.java, MostraNumeriInteriFileBinario.java, MostraNumeriInteriFileDiTesto.java, OrdinaNumeriInteriNoDuplicatiFileDiTesto.java e OrdinaNumeriInteriNoDuplicatiFileBinario.java

**3.**

Si usi il file words.txt

**Soluzione**: Si veda il codice del file sorgente WordPuzzle.java

\*\*\*\*\*\*

**Progetti**

**1.**

Questo progetto appare semplice, anche se non lo è. La definizione del problema è abbastanza chiara, con solo un paio di ambiguità da chiarire. La soluzione qui mostrata presuppone che ci sia almeno una riga nel file da elaborare. Un metodo di supporto elabora la prima parte della prima riga per rimuovere tutti gli spazi bianchi in testa e per impostare in maiuscolo la prima lettera. Dopo di che, il metodo main legge semplicemente una riga di testo alla volta fino a quando non ottiene una stringa nulla, e usa un altro metodo di supporto per elaborare il testo rimanente. Questo metodo di supporto è quello in cui viene svolta la maggior parte del lavoro: elabora ogni riga, carattere per carattere, e usa dei flag per controllare l'elaborazione, per stampare solo uno spazio quando ce n'è più di uno in sequenza, e per capitalizzare la prima parola in ogni frase. Si noti come viene scritto il metodo di supporto per convertire un carattere in maiuscolo. Il codice controlla prima di tutto se il carattere è una lettera minuscola, e, se lo è, converte il codice minuscolo ASCII nel corrispondente codice maiuscolo ASCII. Un controllo sulla tabella ASCII mostrerà che i codici maiuscoli sono inferiori di 32 unità rispetto a quelli delle corrispondenti lettere minuscole. Dopo aver fatto l’opportuna sottrazione, è necessario riportare il risultato intero a char per far corrispondere il tipo di ritorno del metodo.

**Riferimenti**: Programma 1 Capitolo 14 e Programma 2 Capitolo 14

**Soluzione**: Si veda il codice dei file sorgente ScriviFraseFileDiTesto.java, MostraFraseFileDiTesto.java e EditaFraseFileDiTesto.java

**2.**

Questo progetto richiede un'attenta collocazione dei blocchi try/catch per catturare i casi in cui i nomi di file o le specie non esistono e allo steso tempo permettere all’utente di continuare ad usare il programma.

Usare il file Specie.java del Listato 14.10.

**Riferimenti**: Listato 14.10, Listato 14.11.

**Soluzione**: Si veda il codice dei file sorgente ScriviSpecieSuFile.java, TrovaSpecie.java, e MostraSpecie.java

**3.**

Questo progetto richiede di implementare un solo file e si appoggia ai sorgenti generati per il progetto 2 del capitolo 14.

Più specie possono avare la stessa popolazione massima e minima. Se più di una specie ha la popolazione più piccola o più grande, la soluzione mostra semplicemente il record della prima popolazione che incontra leggendo il file. I file che includono i record delle specie dovrebbero includere le seguenti situazioni in modo da verificare la correttezza della classe implementata:

* record multipli con le popolazioni di dimensioni più piccole e più grandi,
* popolazioni con dimensioni non in ordine crescente o decrescente,
* record con valori più grandi e più piccoli della popolazione in varie posizioni del file (il primo record, l'ultimo record e una posizione intermedia),
* il record con la popolazione più piccola posta prima di quella con la più grande,
* il record con la popolazione più numerosa posta prima di quella con la più piccola.

Usare il file Specie.java del Listato 14.10 e ScriviSpecieSuFile.java del Progetto 2, Capitolo 14.

**Riferimenti**: Progetto 2, Capitolo 14, Listato 14.10, Listato 14.11.

**Soluzione**: Si veda il codice del file sorgente SpeciePopolazioniMaggioreMinore.java

**4.**

Questo progetto richiede di implementare un solo file e si appoggia ai sorgenti generati per il progetto 2 del capitolo 14.

La soluzione proposta prevede quattro chiamate di metodo nel main: una per aprire un file di input per la lettura di record di specie, una per aprire un file di output per scrivere nuovi record, una per fare l'elaborazione (leggere un record dal file di input e scrivere un record nel file di output con la popolazione cambiata con il valore proiettato dopo 100 anni), e una per chiudere i file di input e di output.

Usare il file Specie.java del Listato 14.10 e il file generato da ScriviSpecieSuFile.java del Progetto 2, Capitolo 14.

**Riferimenti**: Progetto 2, Capitolo 14, Listato 14.10, Listato 14.11.

**Riferimenti**: Progetto 3, capitolo 11.

**Soluzione**: Si veda il codice del file sorgente SpecieFra100Anni.java

**5.**

La soluzione per questo progetto si avvale di una coppia di metodi. Il metodo principale elabora e contrassegna una linea che contiene una abbreviazione. Trova l'indice dell'abbreviazione nella linea e poi suddivide la linea in 3 parti. Ottiene quindi il carattere immediatamente prima e dopo l'abbreviazione e controlla se si tratta di una lettera o di una cifra. Se è così, allora si suppone che l'abbreviazione sia parte di una parola e non marca quindi la linea. Altrimenti, verrà utilizzato il marcatore <>. Questo viene fatto in un ciclo che elabora la parte rimanente della linea fino a quando l'abbreviazione non viene trovata.

Usare il file abbreviazioni.txt

**Soluzione**: Si veda il codice del file sorgente DelimitatoreDiAbbreviazioni.java

**6.**

Una difficoltà nella soluzione di questo progetto è che non sono ancora state affrontate le collezioni. Verranno letti i numeri di telefono ed inseriti in un array. Occorre determinare il numero di oggetti nel file oppure occorre espandere l'array mentre leggiamo gli oggetti dal file nel caso questo non sia capiente a sufficienza. Questa soluzione legge il file due volte, una volta per determinare il numero di oggetti e poi per leggere gli oggetti ed inserirli in un array di dimensione adeguata.

Si usi il file numeri.data

**Riferimenti**: Esercizio 6 Capitolo 14.

**Soluzione**: Si veda il codice dei file sorgente NumeroDiTelefonoSerializzato.java, NumeroDiTelefonoSerializzatoDemo.java, e FileNumeriTelefoniMancanteException.java

**7.**

Questo progetto può essere risolto modificando la classe IOOggettiClasseDemo, Listato 14.11. La classe PetRecord non possiede i metodi readInput() e toString() come la classe Specie, di conseguenza sono stati aggiunti. Si noti inoltre che PetRecord deve implementare Serializable.

**Riferimenti**: Listato 9.1, Listato 14.10 e Listato 14.11.

**Soluzione**: Si veda il codice dei file sorgente AnimaleRecord.java e AnimaleRecordLeggiScrivi.java

**8.**

**Riferimenti**: Listato 9.1 e Listato 14.11.

**Soluzione**: Si veda il codice del file sorgente AnimaliDemoPesiEta.java

**9.**

Il caso di studio del Listato 14.5 mostra come elaborare un file CSV e quindi risulta essere un buon punto di partenza per la risoluzione di questo progetto.

Si usino i file haberman.names, haberman.data.

**Riferimenti**: Listato 14.5.

**Soluzione**: Si veda il codice del file sorgente Haberman.java

**12.**

Il codice segue fedelmente l’esempio fornito nel testo per la classe URL. Al fine di completarlo, occorre inserire un blocco try-catch e di includere

**Riferimenti**: Listato 14.5.

**Soluzione**: Si veda il codice del file sorgente Browser.java