ESERCIZI DEL MANUALE DI JAVA 9

(http://www.hoeplieditore.it/8302-2)

Indice degli esercizi e delle soluzioni

Introduzione agli esercizi

Questo documento rappresenta, insieme al documento delle appendici, il naturale completamento dell'opera "Manuale di Java 9".
Tutti gli esercizi sono stati spostati in questo documento per non togliere spazio alla teoria. Questo perché per precisa volontà dell'editore, si è voluto contenere nell'edizione testuale il numero di pagine supplementari rispetto a quello dell'edizione precedente "Manuale di Java 8". Per ogni capitolo del libro (e per ogni appendice, dove applicabile) sono stati organizzati esercizi specifici per validare (ed ampliare) quanto appreso durante lo studio.
Gli esercizi sono fondamentali, imprescindibili. Infatti la teoria solitamente risulta chiara, ma applicare i concetti appresi è tutt'altro che facile. La programmazione non comprende solo l'implementazione del codice, bensì molte altre componenti ne fanno parte e influiscono sul risultato finale.
Gli esercizi che troverete in questo documento quindi si sforzano di insistere anche su argomenti che altri libri trattano superficialmente, o non trattano per niente, come l'analisi, la progettazione, l'architettura object oriented. Inoltre esistono esercizi per tutti gli argomenti, anche per quelli che possono sembrare banali al lettore più esperto, ma che possono risultare fondamentali per il neofita. In questa edizione, il numero degli esercizi è stato più che raddoppiato rispetto alla versione precedente.

Attenzione! Questo documento è stato aggiornato a maggio 2018. Questa versione copre tutti i capitoli del libro, più le appendici, per un totale di oltre 400 esercizi!

L'autore si riserva di notificare qualsiasi novità o informazione, tramite i suoi canali social.

In base ai feedback dei lettori, abbiamo riorganizzato e migliorato la chiarezza, la forma espositiva, e creato esercizi che possono soddisfare tutte le tipologie di lettori. Sono stati introdotti anche centinaia di esercizi a risposta singola o multipla, sulla falsariga dei test per la certificazione OCA (Oracle Certification Associate). Tali esercizi supportano la preparazione alla certificazione, e sono stati impostati in modo tale da far acquisire sicurezza al programmatore. Spesso consistono essenzialmente nel leggere del codice e comprenderne il significato nei dettagli, e rappresentano anche per i più esperti un banco di prova notevole. Siamo sicuri che il lettore apprezzerà lo sforzo compiuto per realizzare un così gran numero di esercizi eterogenei e tutti originali.
Diversamente da molti altri testi, abbiamo comunque fornito delle soluzioni per tutti gli esercizi (con poche eccezioni). Ovviamente quando si tratta di codificare una soluzione, esistono centinaia di valide alternative, quindi non bisogna considerare le soluzioni proposte come quelle oggettivamente migliori. È possibile scaricare tutti i listati delle soluzioni e degli esercizi (insieme a tutti gli esempi di codice inclusi nel testo) all'indirizzo: http://www.claudiodesio.com/java9.html.
Ricordiamo ancora una volta che, soprattutto per chi è alle prime armi, è importante iniziare a scrivere tutto il codice a mano, senza copia-incolla o particolari aiuti da parte dello strumento di sviluppo che si sta utilizzando. È anche molto importante commentare il proprio codice, soprattutto all'inizio riga per riga. Questo vi permetterà di fissare bene le definizioni, e ad avere maggiore sicurezza nello scrivere codice.
Sarà sufficiente scrivere il codice sorgente su un editor di testo come Blocco Note di Windows (come descritto nel capitolo 1) e compilare da riga di comando (si raccomanda la lettura dell'appendice A).
Sconsigliamo di svolgere gli esercizi di questi primissimi capitoli (diciamo i primi 4) utilizzando un IDE complesso come Eclipse o Netbeans... si potrebbe finire a studiare l'IDE piuttosto che Java.
È invece consigliato (se non si vuole avere troppo a che fare con Blocco Note e la riga di comando) l'utilizzo di EJE (https://sourceforge.net/projects/eje) che offre delle semplici utility pensate proprio per chi inizia a programmare. Per esempio consente di compilare e mandare in esecuzione i nostri file con la pressione di due semplici pulsanti.

EJE è di semplice installazione (basta decomprimere il file in una cartella qualsiasi) ed utilizzo. Ad ogni modo l'appendice V è dedicata alla sua descrizione.

Infine suggeriamo, dopo aver svolto un esercizio, di consultare la soluzione relativa prima di passare al successivo (spesso c'è propedeuticità tra un esercizio e il successivo), o almeno di leggere le soluzioni.
Per qualsiasi tipo di segnalazione, è possibile scrivere direttamente all'autore all'indirizzo: claudio@claudiodesio.com. È anche possibile contattarmi tramite i più importanti social network e sul mio sito personale (anche per essere sempre aggiornati sulle novità):

- Facebook: http://www.facebook.com/claudiodesiocesari
- Twitter: http://twitter.com/cdesio
- LinkedIn: http://www.linkedin.com/in/claudiodesio
- Google+: http://plus.google.com/+ClaudioDeSioCesari
- Internet: http://www.claudiodesio.com
- Youtube: https://www.youtube.com/c/claudiodesiocesari
- Instagram: https://www.instagram.com/cdesio

Se siete interessati solo alle notizie e alle novità riguardanti questo manuale, potete anche iscrivervi al canale Telegram dedicato: "Aggiornamenti su Manuale di Java 9, Claudio De Sio Cesari (Hoepli editore)", all'indirizzo:

- Telegram: https://t.me/java9desio

Buon lavoro!

Claudio De Sio Cesari

Esercizi del capitolo 1

Introduzione a Java

I seguenti esercizi sono stati pensati per chi inizia da zero. Questi hanno l'unico scopo di far prendere un minimo di confidenza con l'ambiente di programmazione Java. Ricordiamo ancora una volta che, soprattutto se si sta iniziando, è importante iniziare a scrivere tutto il codice a mano, senza copia incolla, o particolari aiuti da parte del tool di sviluppo che si sta utilizzando.
Sarà sufficiente scrivere il codice sorgente su un editor di testo come il blocco note di Windows (come descritto nel capitolo 1) e compilare da riga di comando (si raccomanda la lettura dell'appendice A).
Sconsigliamo di svolgere gli esercizi di questi primissimi capitoli (diciamo i primi 4) utilizzando un IDE complesso come Eclipse o Netbeans... si potrebbe finire a studiare l'IDE piuttosto che Java.
È invece consigliato (se non si vuole avere troppo a che fare con il blocco note e la riga di comando) l'utilizzo di EJE, che offre delle semplici utility pensate proprio per chi inizia a programmare. Per esempio permette di compilare e mandare in esecuzione i nostri file con la pressione di due semplici pulsanti.

EJE si deve scaricare all'indirizzo:
https://sourceforge.net/projects/eje. È di semplice installazione (basta decomprimere il file in una cartella qualsiasi). Per eseguirlo bisogna fare un doppio clic sul file eje.bat. L'appendice V è dedicata alla sua descrizione.

Dal capitolo 5 in poi sarà più semplice passare ad un IDE, visto che saranno spiegati altri ambienti di sviluppo. Infine, dopo aver svolto un esercizio, suggeriamo di consultare la soluzione relativa prima di passare al successivo (spesso c'è propedeuticità tra un esercizio e il successivo). Questo consiglio vale per tutti gli esercizi di tutti i capitoli e appendici. Allo stesso indirizzo dove avete scaricato queste appendici (http://www.claudiodesio.com/java9.html) potete scaricare anche tutto il codice di esempio contenuto nel libro e nelle appendici (sempre scaricabili allo stesso indirizzo) e tutti i listati degli esercizi relativi a tutti i capitoli e tutte le appendici.

Esercizio 1.a)

Digitare, salvare, compilare ed eseguire il programma `HelloWorld`. Consigliamo al lettore di eseguire questo esercizio due volte: la prima volta utilizzando il Notepad e la prompt DOS, e la seconda utilizzando EJE.

> **EJE permette di inserire parti di codice pre-formattate tramite il menu Inserisci (o tramite short-cut).**

Esercizio 1.b) Concetti base di informatica, Vero o Falso:

1. Un computer è composto da hardware e software.
2. Il sistema operativo fa parte dell'hardware di un computer. Infatti un computer non può funzionare senza sistema operativo.
3. Blocco Note di Windows è un software.
4. Il cavo dell'alimentatore di un computer è un hardware.
5. Il linguaggio macchina è il linguaggio che il processore di un computer riesce ad interpretare.
6. Il linguaggio macchina è unico e standard.
7. Il linguaggio macchina ha un vocabolario formato solo da due simboli `0` e `1`.
8. Sia il compilatore che l'interprete hanno il compito di tradurre le istruzioni scritte con un certo linguaggio di programmazione in istruzioni in linguaggio macchina.
9. In generale, un programma scritto in un linguaggio interpretato ha un tempo di esecuzione (runtime) dei programmi più veloce rispetto ad un programma scritto con un linguaggio compilato.

10. Un programma eseguibile è composto dai suoi file sorgente.

Esercizio 1.c) Caratteristiche di Java, Vero o Falso:

1. Java è il nome di una tecnologia e contemporaneamente il nome di un linguaggio di programmazione.

2. Java è un linguaggio interpretato ma non compilato.

3. Java è un linguaggio veloce ma non robusto.

4. Java è un linguaggio difficile da imparare perché in ogni caso obbliga ad imparare l'Object Orientation.

5. La Java Virtual Machine è un software che supervisiona il software scritto in Java.

6. La JVM gestisce la memoria automaticamente mediante la Garbage Collection.

7. L'indipendenza dalla piattaforma è una caratteristica poco importante.

8. Java è un sistema chiuso.

9. La Garbage Collection garantisce l'indipendenza dalla piattaforma.

10. Java è un linguaggio gratuito che raccoglie le caratteristiche migliori di altri linguaggi, e ne esclude quelle ritenute peggiori e più pericolose.

Esercizio 1.d) Codice Java, Vero o Falso:

1. La seguente dichiarazione del metodo `main()` è corretta:

```
public static main(String argomenti[]) {...}
```

2. La seguente dichiarazione del metodo `main()` è corretta:

```
public static void Main(String args[]){...}
```

3. La seguente dichiarazione del metodo main() è corretta:

```
public static void main(String argomenti[]) {...}
```

4. La seguente dichiarazione del metodo main() è corretta:

```
public static void main(String Argomenti[]) {...}
```

5. La seguente dichiarazione di classe è corretta:

```
public class {...}
```

6. La seguente dichiarazione di classe è corretta:

```
public Class Auto {...}
```

7. La seguente dichiarazione di classe è corretta:

```
public class Auto {...}
```

8. È possibile dichiarare un metodo al di fuori del blocco di codice che definisce una classe.

9. Il blocco di codice che definisce un metodo è delimitato da due parentesi tonde.

10. Il blocco di codice che definisce un metodo è delimitato da due parentesi quadre.

Esercizio 1.e) Ambiente e processo di sviluppo, Vero o Falso:

1. La JVM è un software che simula un hardware.

2. Il bytecode è contenuto in un file con suffisso ".class".

3. Lo sviluppo Java consiste nello scrivere il programma, salvarlo, mandarlo in esecuzione ed infine compilarlo.

4. Lo sviluppo Java consiste nello scrivere il programma, salvarlo, compilarlo ed infine mandarlo in esecuzione.

5. Il nome del file che contiene una classe Java deve coincidere con il nome della classe, anche se non si tiene conto delle lettere maiuscole e minuscole.

6. Una volta compilato un programma scritto in Java è possibile eseguirlo su di un qualsiasi sistema operativo che abbia una JVM.

7. Per eseguire una qualsiasi applicazione Java basta avere un browser.

8. Il compilatore del JDK viene invocato tramite il comando javac e la JVM viene invocata tramite il comando java.

9. Per mandare in esecuzione un file che si chiama Pippo.class, dobbiamo eseguire il seguente comando dalla prompt: java Pippo.java.

10. Per mandare in esecuzione un file che si chiama Pippo.class, dobbiamo eseguire il seguente comando dalla prompt: java Pippo.class.

Esercizio 1.f)

Eliminare il modificatore `static` del metodo `main()` dalla classe `HelloWorld`. Utilizzando la prompt DOS, compilare ed eseguire il programma, e interpretare il messaggio di errore dell'esecuzione.

Saper interpretare i messaggi di errore è assolutamente fondamentale.

Esercizio 1.g)

Eliminare la prima parentesi graffa aperta incontrata dalla classe `HelloWorld`. Utilizzando la prompt DOS, compilare il programma e interpretare il messaggio di errore della compilazione.

Esercizio 1.h)

Eliminare l'ultima parentesi chiusa (ultimo simbolo del programma) dalla classe `HelloWorld`. Utilizzando la prompt DOS, compilare il programma e interpretare il messaggio di errore.

Esercizio 1.i)

Eliminare il simbolo di `;` dalla classe `HelloWorld`. Utilizzando la prompt DOS compilare il programma e interpretare il messaggio di errore.

Esercizio 1.l)

Raddoppiare il simbolo di `;` nel programma `HelloWorld` compilare ed eseguire, cosa succede?

Esercizio 1.m)

Scrivere il programma `HelloWorld` scrivendo ogni parola e ogni simbolo al rigo successivo.

Esercizio 1.n)

Provare a far stampare una stringa a piacere al programma `HelloWorld` al posto della stringa `Hello World!`.

Esercizio 1.o)

Provare a far stampare un numero al programma `HelloWorld` al posto della stringa `Hello World!`.

Esercizio 1.p)

Provare a far stampare la somma di due numeri al programma `HelloWorld` al posto della stringa `Hello World!`, dopo aver letto la soluzione dell'esercizio precedente 1.o.

Esercizio 1.q)

Compilare ed eseguire il seguente programma:

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

    }
}
```

cosa viene stampato?

Esercizio 1.r)

Compilare ed eseguire il seguente programma:

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("");
    }
}
```

cosa viene stampato?

Esercizio 1.s)

Compilare ed eseguire il seguente programma:

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println(args);
    }
}
```

cosa viene stampato?

Esercizio 1.t)

Scrivere il programma `Elenco` che scrive un elenco della spesa, dove ogni articolo da comprare risiede sul proprio rigo.

Esercizio 1.u)

Scrivere il programma `Elenco` che scrive un elenco della spesa, dove ogni articolo è separato da un altro con una virgola.

Esercizio 1.v)

Creare un nuovo file di nome `SayJava` che stampa il vostro nome come nel seguente esempio:

```
--------
| JAVA |
--------
```

Utilizzare solo la riga di comando (prompt DOS) e Blocco Note.

Esercizio 1.z)

Creare un nuovo file di nome `SayMyName` che stampa il vostro nome come nel seguente esempio (si noti che sono state utilizzate 5 colonne e 5 righe):

```
*****  *       ******   *   *   **      *   *****
*      *       *    *   *   *  *  *     *   *   *
*      *       ******   *   *  *    *    *   *   *
*      *       *    *   *   *  *  *     *   *   *
*****  *****   *    *   *****  ***      *   *****
```

Utilizzare solo la riga di comando (prompt DOS) e Blocco Note.

Soluzioni degli esercizi del capitolo 1

Le soluzioni di codice proposte per questo e tutti gli altri capitoli, non rappresentano le uniche soluzioni esistenti in assoluto. Per risolvere i nostri esercizi, esistono spesso centinaia di soluzioni diverse e tutte valide. Ognuna di essa ha dei pro e dei contro. Quindi non bisogna prendere la soluzione proposta per un esercizio come l'unica esistente. Questo concetto vale anche per tutti gli altri capitoli. Questo non si applica a soluzioni non costituite da codice (per esempio esercizi "Vero o Falso").

Soluzione 1.a)

Il listato è simile al seguente:

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}
```

Per le operazioni da eseguire dalla prompt DOS per compilare ed eseguire il codice consultare i paragrafi 1.3.2 e 1.4.3.

Soluzione 1.b) Concetti base di informatica, Vero o Falso:

1. **Vero**.
2. **Falso**, il sistema operativo è un software.
3. **Vero**.

4. **Vero**.
5. **Vero**.
6. **Falso**, ogni processore conosce il suo codice sorgente.
7. **Vero**.
8. **Vero**.
9. **Falso**, perché un linguaggio interpretato deve alternare alla fase di esecuzione la fase di traduzione, quindi è solitamente più lento.
10. **Falso**, un programma eseguibile è composto dai suoi file binari.

Soluzione 1.c) Caratteristiche di Java, Vero o Falso:

1. **Vero**.
2. **Falso**.
3. **Falso**.
4. **Vero**.
5. **Vero**.
6. **Vero**.
7. **Falso**.
8. **Falso**.
9. **Falso**.
10. **Vero**.

Soluzione 1.d) Codice Java, Vero o Falso:

1. **Falso**, manca il tipo di ritorno (`void`).
2. **Falso**, l'identificatore dovrebbe iniziare con lettera minuscola (`main`).
3. **Vero**.
4. **Vero**.
5. **Falso**, manca l'identificatore.

6. **Falso**, la parola chiave si scrive con lettera iniziale minuscola (`class`).
7. **Vero**.
8. **Falso**.
9. **Falso**, le parentesi sono graffe.
10. **Falso**, le parentesi sono graffe.

Soluzione 1.e) Ambiente e processo di sviluppo, Vero o Falso:

1. **Vero**.
2. **Vero**.
3. **Falso**, bisogna prima compilarlo per poi mandarlo in esecuzione.
4. **Vero**.
5. **Falso**, bisogna anche tenere conto delle lettere maiuscole e minuscole.
6. **Vero**.
7. **Falso**, un browser è sufficiente solo per eseguire applet.
8. **Vero**.
9. **Falso**, il comando giusto è java Pippo.
10. **Falso**, il comando giusto è java Pippo.

Soluzione 1.f)

Il listato dovrebbe essere simile al seguente:

```
public class HelloWorld {
    public void main(String args[]) {
        System.out.println("Hello World!");
    }
}
```

A seconda dell'errore, del vostro sistema operativo (o della virtual machine installata), i messaggi del compilatore potrebbero essere in italiano o in inglese. Se il messaggio è in italiano dovrebbe essere il seguente:

```
Errore: il metodo principale non è static nella classe HelloWorld.
  Definire il metodo principale come:
   public static void main(String[] args)
```

Nel caso i messaggi fossero in inglese si tenga presente che nel campo informatico un minimo di inglese tecnico lo si deve conoscere.

Soluzione 1.g)

Il listato dovrebbe essere simile al seguente:

```
public class HelloWorld
    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}
```

Il messaggio di errore del compilatore (questa volta in inglese) è il seguente:

```
error: '{' expected
public class HelloWorld
                     ^
1 error
```

e non sembra molto complicato.

Soluzione 1.h)

Il listato dovrebbe essere simile al seguente:

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
```

Il messaggio di errore del compilatore in inglese è il seguente:

```
error: reached end of file while parsing
    }
     ^
1 error
```

che ci dice che è stata raggiunta la fine del file... e manca qualcosa.

Soluzione 1.i)

Il listato dovrebbe essere simile al seguente:

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World!")
    }
}
```

Il messaggio di errore del compilatore in inglese è il seguente:

```
error: ';' expected
        System.out.println("Hello World!")
                                           ^
1 error
```

Dove il compilatore ci avverte che manca un punto e virgola.

Soluzione 1.l)

Il listato dovrebbe essere simile al seguente:

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World!");;
    }
}
```

Il file però viene compilato ed eseguito senza errori. Infatti il simbolo `;` superfluo, viene considerato dal compilatore come una terminazione (legale) di uno statement vuoto. Potremmo scriverlo anche al rigo successivo (visto che come abbiamo letto nel capitolo 1 non cambia nulla) per rendere l'idea un po' più chiaramente:

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World!");
        ;
    }
}
```

Soluzione 1.m)

Il listato dovrebbe essere simile al seguente:

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

```
)
{
System
.
out
.
println
(
"Hello World!"
)
;
}
}
```

Il file però viene compilato ed eseguito senza errori.

Soluzione 1.n)

Il listato potrebbe essere simile al seguente:

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Una frase a piacere!");
    }
}
```

Soluzione 1.o)

Il listato potrebbe essere simile al seguente:

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("8");
    }
}
```

Ma abbiamo stampato il numero come stringa (le stringhe saranno argomento del terzo capitolo), infatti l'abbiamo rinchiuso tra due virgolette. Potremmo scrivere anche direttamente:

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println(8);
    }
}
```

Questa volta stiamo stampando un tipo di dato diverso (non ci sono virgolette). Nel prossimo esercizio inizieremo a capire meglio la situazione.

Soluzione 1.p)

Il listato potrebbe essere simile al seguente:

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("25+7");
    }
}
```

L'output quindi non stamperà una somma ma semplicemente:

```
25+7
```

Potremmo scrivere anche questo direttamente:

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println(25+7);
    }
}
```

In questo caso l'output sarà quello desiderato:

```
32
```

Infatti i tipi di dati numerici (che vengono scritti senza virgolette come vedremo nel capitolo 3) ci permettono di eseguire operazioni aritmetiche.

Soluzione 1.q)

Il programma non stampa niente perché manca l'istruzione di stampa.

Soluzione 1.r)

Il programma non stampa niente perché all'istruzione di stampa non viene passato nulla da stampare. Si può notare però che il cursore è sceso alla riga successiva, per effetto del fatto che l'istruzione `System.out.println()` va sempre a capo dopo aver stampato (ma anche dopo non aver stampato). Infatti `println`, sta per "print line" (in italiano "stampa riga").

Soluzione 1.s)

In questo caso l'output sarà simile al seguente:

```
[Ljava.lang.String;@5679c6c6
```

È stato "stampato un oggetto" (chiamato `args`) e capiremo nei prossimi capitoli il perché abbia una rappresentazione sotto forma di stringa così misteriosa.

Soluzione 1.t)

Il listato dovrebbe essere simile al seguente:

```
public class Elenco {
    public static void main(String args[]) {
        System.out.println("Pane, caffè, tè, frutta");
    }
}
```

E l'output è il seguente:

```
Pane, caffè, tè, frutta
```

Soluzione 1.u)

Il listato dovrebbe essere simile al seguente:

```
public class Elenco {
    public static void main(String args[]) {
        System.out.println("Pane");
        System.out.println("caffè");
        System.out.println("tè");
        System.out.println("frutta");
    }
}
```

E l'output è il seguente:

```
Pane
caffè
tè
frutta
```

Soluzione 1.v)

Il listato potrebbe essere simile al seguente:

```
public class SayJava {
    public static void main(String args[]) {
        System.out.println("--------");
        System.out.println("| JAVA |");
        System.out.println("--------");
    }
}
```

Soluzione 1.z)

Il listato potrebbe essere simile al seguente:

```
public class SayMyName {
    public static void main(String args[]) {
        System.out.println("*****  *      ******   *   *   **     *   *****");
        System.out.println("*      *      *    *   *   *   *  *    *   *   *");
        System.out.println("*      *      ******   *   *   *   *   *   *   *");
        System.out.println("*      *      *    *   *   *   *  *    *   *   *");
        System.out.println("*****  *****  *    *   *****   ***     *   *****");
    }
}
```

Esercizi del capitolo 2

Componenti fondamentali di un programma Java

Ricordiamo che molti esercizi sono propedeutici ai successivi, quindi non conviene saltarne qualcuno o quantomeno vi consigliamo di consultare le soluzioni prima di andare avanti.

Esercizio 2.a)

Viene fornita (copiare, salvare e compilare) la seguente classe:

```
public class NumeroIntero {
    public int numeroIntero;
    public void stampaNumero() {
        System.out.println(numeroIntero);
    }
}
```

Questa classe definisce il concetto di numero intero come oggetto. In essa vengono dichiarati una variabile intera ed un metodo che stamperà la variabile stessa. Scrivere, compilare ed eseguire una classe che:

- istanzierà almeno due oggetti dalla classe `NumeroIntero` (contenente un metodo `main()`);
- cambierà il valore delle relative variabili e testerà la veridicità delle avvenute assegnazioni, sfruttando il metodo `stampaNumero()`;

- aggiungerà un costruttore alla classe `NumeroIntero` che inizializzi la variabile d'istanza.

Due domande ancora:

1. a che tipologia di variabili appartiene la variabile `numeroIntero` definita nella classe `NumeroIntero`?
2. Se istanziamo un oggetto della classe `NumeroIntero`, senza assegnare un nuovo valore alla variabile `numeroIntero`, quanto varrà quest'ultima?

Esercizio 2.b) Concetti sui componenti fondamentali, Vero o Falso:

1. Una variabile d'istanza deve essere per forza inizializzata dal programmatore.
2. Una variabile locale condivide il ciclo di vita con l'oggetto in cui è definita.
3. Un parametro ha un ciclo di vita coincidente con il metodo in cui è dichiarato: nasce quando il metodo viene invocato, muore quando termina il metodo.
4. Una variabile d'istanza appartiene alla classe in cui è dichiarata.
5. Un metodo è sinonimo di azione, operazione.
6. Sia le variabili sia i metodi sono utilizzabili di solito mediante l'operatore dot, applicato ad un'istanza della classe dove sono stati dichiarati.
7. Un costruttore è un metodo che non restituisce mai niente, infatti ha come tipo di ritorno `void`.
8. Un costruttore viene detto di default, se non ha parametri.
9. Un costruttore è un metodo e quindi può essere utilizzato mediante l'operatore dot, applicato ad un'istanza della classe dove è stato dichiarato.
10. Un package è fisicamente una cartella che contiene classi, le quali hanno dichiarato esplicitamente di far parte del package stesso nei rispettivi file sorgente.

Esercizio 2.c) Sintassi dei componenti fondamentali. Vero o Falso:

1. Nella dichiarazione di un metodo, il nome è sempre seguito dalle parentesi che circondano i parametri opzionali, ed è sempre preceduto da un tipo di ritorno.

2. Il seguente metodo è dichiarato in maniera corretta:

```
public void metodo ()  {
    return 5;
}
```

3. Il seguente metodo è dichiarato in maniera corretta:

```
public int metodo () {
    System.out.println("Ciao");
}
```

4. La seguente variabile è dichiarata in maniera corretta:

```
public int a = 0;
```

5. La seguente variabile `x` è utilizzata in maniera corretta (fare riferimento alla classe `Punto` definita in questo capitolo):

```
Punto p1 = new Punto();
Punto.x = 10;
```

6. La seguente variabile `x` è utilizzata in maniera corretta (fare riferimento alla classe `Punto` definita in questo capitolo):

```
Punto p1 = new Punto();
Punto.p1.x = 10;
```

7. La seguente variabile `x` è utilizzata in maniera corretta (fare riferimento alla classe `Punto` definita in questo capitolo):

```
Punto p1 = new Punto();
x = 10;
```

8. Il seguente costruttore è utilizzato in maniera corretta (fare riferimento alla classe `Punto` definita in questo capitolo):

```
Punto p1 = new Punto();
p1.Punto();
```

9. Il seguente costruttore è dichiarato in maniera corretta:

```
public class Computer {
    public void Computer(){
    }
}
```

10. Il seguente costruttore è dichiarato in maniera corretta:

```
public class Computer {
    public computer(int a) {

    }
}
```

Esercizio 2.d)

Creare una classe `Quadrato`, che dichiari una variabile d'istanza intera `lato`. Quindi creare un metodo pubblico che si chiami `perimetro()` e che ritorni il perimetro del quadrato, e un metodo pubblico `area()` che ritorni l'area del quadrato.

Ricordiamo per chi lo ha dimenticato, che il perimetro è la somma dei lati del quadrato, mentre l'area si calcola moltiplicando il lato per se stesso. Infine il simbolo per eseguire una moltiplicazione in Java è il `*`.

Esercizio 2.e)

Creare una classe `TestQuadrato` che contenga un metodo `main()` che istanzi un oggetto di tipo `Quadrato`, con lato di valore `5`. Quindi stampare il perimetro e l'area dell'oggetto appena creato.

Esercizio 2.f)

Dopo aver svolto l'esercizio precedente, dovreste aver impostato la variabile `lato` a mano con un'istruzione come la seguente:

```
nomeOggetto.lato = 5;
```

Per poter evitare questa istruzione si crei un costruttore nella classe `Quadrato` dell'esercizio 2.d, che prenda in input il valore della variabile `lato`. Dopo aver finito, si compili la classe `Quadrato`. La classe `TestQuadrato` invece, non compilerà più per via dell'istruzione specificata sopra e per il mancato utilizzo del nuovo costruttore. Si modifichi il codice della classe `TestQuadrato` in modo tale che compili e sia eseguita correttamente.

Esercizio 2.g)

Nella classe `Quadrato` creata nell'esercizio 2.d, sostituire il valore 4 usato per calcolare il perimetro, con una costante d'istanza `NUMERO_LATI`.

> **Si noti che per la costante è stato usato un nome costituito da sole lettere maiuscole separate con un simbolo di sottolineatura (underscore). Questa è una convenzione che si usa per le costanti come spiegato nel paragrafo 3.1.**

Questo non dovrebbe influire sulla classe `TestQuadrato`.

Esercizio 2.h)

Creare una classe `Rettangolo`, equivalente alla classe `Quadrato` creata nell'esercizio 2.d e perfezionata negli esercizi successivi. Prima di codificare la classe decidere che specifiche deve avere questa classe (variabili e metodi).

> **Mai buttarsi sul codice direttamente. È un errore classico che soprattutto all'inizio può portare a perdersi. Bisogna prima definire nella propria mente, o ancora meglio su un foglio di carta, le specifiche. Il consiglio è quello di avere ben chiare le varie definizioni (variabili d'istanza, variabili locali, metodi, costruttori, etc.).**

Esercizio 2.i)

Creare una classe `TestRettangolo` che contenga un metodo `main()` e che testi la classe `Rettangolo`, equivalentemente a come abbiamo fatto nell'esercizio 2.f. Questa volta si istanzino almeno due rettangoli diversi.

Esercizio 2.l)

Si astragga con una classe il concetto di `Nazione`, creando almeno un costruttore e delle variabili d'istanza, ma nessun metodo.

Esercizio 2.m)

Dopo aver creato la classe `Nazione` dell'esercizio 2.l, riuscite a creare uno o più metodi? Se ci riuscite, definiteli all'interno della classe. Se non ci riuscite, sapete spiegare il perché?

Esercizio 2.n)

Data la seguente classe:

```
public class Esercizio2N {
    public String string;
    public int intero;
    final public String integer = "inzializzazione";
}
```

Quale delle seguenti affermazioni è vera (scegliere una sola affermazione)?

1. Il codice può essere compilato correttamente.
2. Il codice non può essere compilato correttamente perché non è possibile dichiarare una variabile con nome `string`.
3. Il codice non può essere compilato correttamente perché non è possibile dichiarare una variabile con nome `intero`.
4. Il codice non può essere compilato correttamente perché non è possibile dichiarare una variabile di tipo `String` chiamandola `integer`.
5. Il codice non può essere compilato correttamente perché la variabile con nome `integer` dichiara i modificatori in ordine inverso (dovrebbe essere prima dichiarata `public` e poi `final`).

Esercizio 2.o)

Data la seguente classe:

```
public class Esercizio2O {
    public String toString() {
        return "Esercizio2O"
    }

    public void main() {

    }

    public void static metodo() {

    }

    static public void main(String argomenti[]) {

    }
}
```

C'è un solo errore in questa classe che ne impedirà la compilazione, quale?

Se non si è in grado di rispondere alla domanda, scrivere e compilare la classe a mano, e interpretare l'errore. Poi risolverlo e verificare la risoluzione compilando.

Esercizio 2.p)

Data la seguente classe:

```
public class Esercizio2P {
    public String stringa;

    public static void metodo(String argz[]) {
        public int intero = 0;
    }
}
```

C'è un solo errore in questa classe che ne impedirà la compilazione, quale?

Se non si è in grado di rispondere alla domanda, scrivere e compilare la classe a mano, e interpretare l'errore. Poi risolverlo e verificare la risoluzione compilando.

Esercizio 2.q)

Data la seguente classe:

```
public class Esercizio2Q {

    public static void main(String args) {
        System.out.println("Ligeia")
    }
  }
}
```

Ci sono ben tre errori che in questa classe ne impediranno la compilazione, quali sono?

Se non si è in grado di rispondere alla domanda, scrivere e compilare la classe a mano, e interpretare l'errore. Poi risolvere gli errori e verificare la risoluzione compilando.

Esercizio 2.r)

Data la seguente classe:

```
public class Esercizio2R {

    public int var1;
    public int var2;

    System.out.println("Esercizio 2.r");

    public Esercizio2R() {

    }

    public Esercizio2R(int a , int b) {
        var1 = b;
        var2 = a;
    }

    public static void main(String args[]) {
        Esercizio2R esercizio2R = new Esercizio2R (4,7);
        System.out.println(esercizio2R.var1);
        System.out.println(esercizio2R.var2);
    }
}
```

Una volta eseguito, cosa stamperà questo programma?

1. Questo programma non può essere eseguito.
2. Questo programma non compila.
3. Stamperà 74.
4. Stamperà 47.

Esercizio 2.s)

Date le seguenti classi:

```
public class Corso {
    public String nome;
```

```
    public Corso() {

    }

    public Corso(String n) {
        nome = n;
    }
}

public class Esercizio2s {

    public static void main(String args[]) {
        Corso corso1 = new Corso();
        corso1.nome = "Java";
        Corso corso2 = new Corso("Java");
        System.out.println(corso1.nome);
        System.out.println(corso2.nome);
    }
}
```

Quale sequenza di istruzioni tra le seguenti serve per eseguire il programma?

1. javac Corso.java, javac Esercizio2S.java, java Corso.
2. javac Corso.java, javac Esercizio2S.java, java Corso.class.
3. javac Corso.java, javac Esercizio2S.java, java Esercizio2S.
4. javac Corso.java, javac Esercizio2S.java, java Esercizio2S Corso.

Esercizio 2.t)

Date le seguenti classi:

```
public class Corso {
    public String nome;

    public Corso() {

    }

    public Corso(String n) {
        nome = n;
    }
}

public class Esercizio2t {

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

```
        Corso corso1 = new Corso();
        corso1.nome = "Java";
        Corso corso2 = new Corso("Java");
        System.out.println(corso1.nome);
        System.out.println(corso2.nome);
    }
}
```

Una volta eseguito, cosa stamperà questo programma?

1. Questo programma non può essere eseguito.
2. Questo programma non compila.
3. Stamperà:

```
Java
Java
```

4. Stamperà:

```
Java
```

Esercizio 2.u)

```
public class Esercizio2u {
    int c = 3;
    public static void main(String args[]) {
        int a = 1;
        int b = 2, c, d = 4;
        System.out.println(a+b+c+d);
    }
}
```

Una volta eseguito, cosa stamperà questo programma?

1. Questo programma non compila.
2. Stamperà “10”.
3. Stamperà: “7”.
4. Stamperà: “0”.

Esercizio 2.v)

Si crei una classe di nome `Esercizio2V` che consenta di ottenere la somma 2, 3, 5 e 10 numeri interi.

Esercizio 2.z)

Creare una classe che astragga il concetto di "città", chiamandola `Citta` (Java non supporta le lettere accentate per i nomi). Dopodiché dichiarare una classe `Nazione` che dichiari una variabile d'istanza `capitale` di tipo `Citta`. Infine creare una classe `Esercizio2Z` che istanzi una `Nazione` con una capitale, e stampi una frase che verifichi l'effettiva associazione tra la nazione e la capitale.

Si consiglia di creare dei costruttori per tali classi.

Soluzioni degli esercizi del capitolo 2

Soluzione 2.a)

Di seguito viene listata una classe che aderisce ai requisiti richiesti:

```
public class ClasseRichiesta {
    public static void main (String args []) {
        NumeroIntero uno = new NumeroIntero();
        NumeroIntero due = new NumeroIntero();
        uno.numeroIntero = 1;
        due.numeroIntero = 2;
        uno.stampaNumero();
        due.stampaNumero();
    }
}
```

Inoltre un costruttore per la classe `NumeroIntero` potrebbe impostare l'unica variabile d'istanza `numeroIntero`:

```
public class NumeroIntero {
    public int numeroIntero;
    public NumeroIntero(int n) {
        numeroIntero = n;
    }
    public void stampaNumero() {
        System.out.println(numeroIntero);
    }
}
```

In tal caso, però, per istanziare oggetti dalla classe `NumeroIntero`, non sarà più possibile utilizzare il costruttore di default (che non sarà più inserito dal compilato-

re). Quindi la seguente istruzione produrrebbe un errore in compilazione:

```
NumeroIntero uno = new NumeroIntero();
```

Bisogna invece creare oggetti passando al costruttore direttamente il valore della variabile da impostare, per esempio:

```
NumeroIntero uno = new NumeroIntero(1);
```

Risposte alle due domande:

1. Trattasi di una variabile d'istanza, perché dichiarata all'interno di una classe, al di fuori di metodi.
2. Il valore sarà zero, ovvero il valore nullo per una variabile intera. Infatti, quando si istanzia un oggetto, le variabili d'istanza vengono inizializzate ai valori nulli se non esplicitamente inizializzate ad altri valori.

Soluzione 2.b) Concetti sui componenti fondamentali. Vero o Falso:

1. **Falso**, una variabile locale deve essere per forza inizializzata dal programmatore.
2. **Falso**, una variabile d'istanza condivide il ciclo di vita con l'oggetto in cui è definita.
3. **Vero**.
4. **Falso**, una variabile d'istanza appartiene ad un oggetto istanziato dalla classe in cui è dichiarata.
5. **Vero**.
6. **Vero**.
7. **Falso**, un costruttore è un metodo che non restituisce mai niente, infatti non ha tipo di ritorno.
8. **Falso**, un costruttore viene detto di default se viene inserito dal compilatore. Inoltre non ha parametri.
9. **Falso**, un costruttore è un metodo speciale che ha la caratteristica di essere invocato una ed una sola volta nel momento in cui si istanzia un oggetto.
10. **Vero**.

Soluzione 2.c) Sintassi dei componenti fondamentali. Vero o Falso:

1. **Vero**.
2. **Falso**, tenta di restituire un valore intero ma possiede tipo di ritorno `void`.
3. **Falso**, il metodo dovrebbe restituire un valore intero.
4. **Vero**.
5. **Falso**, l'operatore dot deve essere applicato all'oggetto e non alla classe:

```
Punto p1 = new Punto();
p1.x = 10;
```

6. **Falso**, l'operatore dot deve essere applicato all'oggetto e non alla classe, inoltre la classe non "contiene" l'oggetto.
7. **Falso**, l'operatore dot deve essere applicato all'oggetto. Il compilatore non troverebbe infatti la dichiarazione della variabile `x`.
8. **Falso**, un costruttore è un metodo speciale che ha la caratteristica di essere invocato una ed una sola volta nel momento in cui si istanzia un oggetto.
9. **Falso**, il costruttore non dichiara tipo di ritorno e deve avere nome coincidente con la classe.
10. **Falso**, il costruttore deve avere nome coincidente con la classe.

Soluzione 2.d)

Il listato dovrebbe essere simile al seguente:

```
public class Quadrato {
    public int lato;

    public int perimetro() {
        int perimetro = lato * 4;
        return perimetro;
    }

    public int area() {
        int area = lato * lato;
        return area;
    }
}
```

Soluzione 2.e)

Il listato dovrebbe essere simile al seguente:

```
public class TestQuadrato {
    public static void main(String args[]) {
        Quadrato quadrato = new Quadrato();
        quadrato.lato = 5;
        int perimetro = quadrato.perimetro();
        System.out.println(perimetro);
        int area = quadrato.area();
        System.out.println(area);
    }
}
```

Si noti che abbiamo creato le variabili locali `perimetro` e `area` con lo stesso nome del metodo, e questo non è un problema. Infatti il nome di un metodo si differenzia sempre dal nome di una variabile perché dichiarato con parentesi tonde. Avremmo anche potuto chiamare in modo diverso le variabili, ma è buona norma che i nomi siano auto-esplicativi. Tuttavia potevamo anche evitare completamente l'uso di queste variabili se avessimo scritto la classe in questo modo:

```
public class TestQuadrato {
    public static void main(String args[]) {
        Quadrato quadrato = new Quadrato();
        quadrato.lato = 5;
        System.out.println(quadrato.perimetro());
        System.out.println(quadrato.area());
    }
}
```

Il codice è più compatto, ma almeno all'inizio conviene utilizzare le variabili per meglio memorizzare le definizioni.

Soluzione 2.f)

Il listato della classe `Quadrato` dovrebbe essere simile al seguente:

```
public class Quadrato {
    public int lato;

    public Quadrato(int l) {
        lato = l;
    }

    public int perimetro() {
        int perimetro = lato * 4;
        return perimetro;
    }
```

```
    public int area() {
        int area = lato * lato;
        return area;
    }
}
```

Il listato della classe `TestQuadrato` modificato dovrebbe invece essere:

```
public class TestQuadrato {
    public static void main(String args[]) {
        Quadrato quadrato = new Quadrato(5);
        int perimetro = quadrato.perimetro();
        System.out.println(perimetro);
        int area = quadrato.area();
        System.out.println(area);
    }
}
```

Soluzione 2.g)

Il listato potrebbe essere simile a:

```
public class Quadrato {
    public final int NUMERO_LATI = 4;

    public int lato;

    public int  perimetro() {
        int perimetro = lato * NUMERO_LATI;
        return perimetro;
    }

    public int area() {
        int area = lato * lato;
        return area;
    }
}
```

Soluzione 2.h)

Il listato per la classe `Rettangolo` potrebbe essere il seguente:

```
public class Rettangolo {
    public final int NUMERO_LATI_UGUALI = 2;
    public int base;
    public int altezza;

    public Rettangolo(int b, int h) {
```

```
        base = b;
        altezza = h;
    }

    public int perimetro() {
        int perimetro = (base + altezza ) * NUMERO_LATI_UGUALI;
        return perimetro;
    }

    public int area() {
        int area = base * altezza;
        return area;
    }
}
```

Soluzione 2.i)

Il listato per la classe `TestRettangolo` potrebbe essere il seguente:

```
public class TestRettangolo {
    public static void main(String args[]) {
        Rettangolo rettangolo1 = new Rettangolo(5,6);
        Rettangolo rettangolo2 = new Rettangolo(8,9);
        System.out.println("Perimetro del rettangolo 1 = "
          + rettangolo1.perimetro());
        System.out.println("Area del rettangolo 1 = "
          + rettangolo1.area());
        System.out.println("Perimetro del rettangolo 2 = "
          + rettangolo2.perimetro());
        System.out.println("Area del rettangolo 2 = "
          + rettangolo2.area());
    }
}
```

Soluzione 2.l)

Il listato per la classe `Nazione` potrebbe essere il seguente:

```
public class Nazione {
    public String nome;
    public String capitale;
    public int popolazione;

    public Nazione(String n, String c, int p) {
        nome = n;
        capitale = c;
        popolazione = p;
    }
}
```

L'astrazione, pur essendo molto generica sembra corretta. L'unico costruttore definito implica la specificazione in fase di istanza dei tre parametri in input, che abbiamo quindi imposto essere obbligatori. Infatti quando istanziamo un oggetto `Nazione`, dobbiamo specificare i parametri di input del costruttore:

```
Nazione italia = new Nazione("Italia", "Roma", "60000000");
```

Non si doveva per forza creare una classe con le stesse variabili, l'importante è avere una propria astrazione corretta.

Soluzione 2.m)

È possibile che qualcuno sia riuscito a creare dei metodi all'interno di questa classe. Nell'esercizio precedente è stata richiesta solo in modo generico un'astrazione della classe `Nazione`, senza specificare il contesto o il programma in cui tale classe dovrà avere un ruolo. È per questo che ci risulta difficile creare dei metodi, visto che attualmente ignoriamo il programma in cui `Nazione` sarà utilizzato. Potremmo utilizzare questa classe in un programma che conserva i dati fisici delle nazioni, ma potremmo anche utilizzarlo in un videogioco che simula il famoso gioco da tavola Risiko. I metodi (ma anche le variabili d'istanza) da definire, potrebbero cambiare drasticamente da contesto a contesto. Nel primo caso infatti definiremmo variabili d'istanza come `fiumi`, `laghi`, `montagne`, `superficie`, etc., e metodi come `produce()`, `esporta()`, `importa()`. Nel secondo caso invece potremmo definire i `confini`, e il metodo `attacca()`.
Concludendo, abbiamo reso estremamente generica la definizione della classe `Nazione`, proprio perché non avevamo vincoli da poter sfruttare.

Soluzione 2.n)

La soluzione è la risposta numero 1, ovvero il codice può essere compilato senza errori. Non ci si faccia trarre in inganno dai nomi delle variabili (vedi altre possibili risposte) che potrebbero portare (volontariamente) a fare confusione. Inoltre non è un problema l'ordine in cui sono specificati i modificatori.

Soluzione 2.o)

L'errore è che manca il `;` accanto allo statement del metodo `toString()`:

```
return "Esempio2O"
```

che dovrebbe essere corretto così:

```
return "Esempio2O";
```

Gli altri metodi sono tutti corretti.

Soluzione 2.p)

L'errore è che non si può dichiarare `public` una variabile locale all'interno di un metodo. Infatti `public` definisce la visibilità al di fuori della classe di una variabile d'istanza, non al di fuori di un metodo.
Abbiamo definito `argz` il parametro del `main()`, in luogo dello standard `args`. Ciò è legale perché si tratta solo di un nome di un parametro.

Soluzione 2.q)

Il primo errore è che mancano le parentesi graffe per il parametro `args` del `main()`. Il secondo è perché manca il `;` accanto all'unico statement del metodo `main()`. Il terzo è dovuto da una parentesi graffa di chiusura in più. Corretti questi errori, la classe compila e può anche essere eseguita visto che contiene un metodo `main()`. La classe corretta sarà la seguente:

```
public class Esercizio2Q_OK {

    public static void main(String args[]) {
        System.out.println("Ligeia");
    }
}
```

Soluzione 2.r)

Il programma non compilerà per via dello statement:

```
System.out.println("Esercizio 2.r");
```

che non appartiene a nessun blocco di codice di metodo. Ma abbiamo visto che all'interno di una classe sono definiti solo variabili e metodi, non statement. Eliminando quello statement, il programma:

```
public class Esercizio2R_OK {

    public int var1;
    public int var2;

    public Esercizio2R_OK() {
```

```
    }

    public Esercizio2R_OK(int a , int b) {
        var1 = b;
        var2 = a;
    }

    public static void main(String args[]) {
        Esercizio2R_OK esercizio2R = new Esercizio2R_OK(4,7);
        System.out.println(esercizio2R.var1);
        System.out.println(esercizio2R.var2);
    }

}
```

stamperà al runtime:

```
7
4
```

Soluzione 2.s)

La risposta corretta è la 3. Sarebbe possibile anche compilare solo la classe `Esercizio2S`, visto che utilizzando la classe `Corso`, essa obbligherà il compilatore a compilare anche quest'ultima. Quindi possiamo anche eseguire questa sequenza:

```
javac Esercizio2S.java
java Esercizio2S
```

Soluzione 2.t)

La risposta corretta è la 3.

Soluzione 2.u)

La risposta corretta è la 1. Infatti la variabile locale `c` non è stata inizializzata e provocherà quest'errore:

```
Esercizio2U.java:6: error: variable c might not have been initialized
        System.out.println(a+b+c+d);
                           ^
1 error
```

Come asserito in questo capitolo, la variabile d'istanza omonima non c'entra nulla con la variabile locale. In qualsiasi caso, una volta inizializzata a `3`:

```
public class Esercizio2u_OK {
    int c = 3;
    public static void main(String args[]) {
        int a = 1;
        int b = 2, c = 3, d = 4;
        System.out.println(a+b+c+d);
    }
}
```

stamperà:

```
10
```

Soluzione 2.v)

Il listato per la classe `Esercizio2V` potrebbe essere il seguente:

```
public class Esercizio2V {

    public int somma2Int(int a, int b) {
        return a+b;
    }

    public int somma5Int(int a, int b, int c, int d, int e) {
        return a+b+c+d+e;
    }


    public int somma10Int(int a, int b, int c, int d, int e,
                          int f, int g, int h, int i, int l) {
        return a+b+c+d+e+f+g+h+i+l;
    }

    //Giusto per testare
    public static void main(String args[]) {
        Esercizio2V es = new Esercizio2V();
        System.out.println(es.somma2Int(1,1));
        System.out.println(es.somma5Int(1,1,1,1,1));
        System.out.println(es.somma10Int(1,1,1,1,1,1,1,1,1,1));
    }
}
```

Non sarebbe del tutto corretto utilizzare un varargs, visto che ci permetterebbe di fare tante altre operazioni che però non sono richieste.

Soluzione 2.z)

Una soluzione potrebbe essere costituita dalla codifica delle seguenti classi:

```
public class Citta {
    public String nome;

    public Citta (String n) {
        nome = n;
    }
}

public class Nazione {
    public String nome;
    public Citta capitale;
    public int popolazione;

    public Nazione (String n, Citta c, int p) {
        nome = n;
        capitale = c;
        popolazione = p;
    }
}

public class Esercizio2z {
    public static void main(String args[]) {
        Citta citta = new Citta("Roma");
        Nazione nazione = new Nazione("Italia", citta, 60000000);
        System.out.println("L'" + nazione.nome
         + " ha come capitale " + citta.nome);
    }
}
```

Esercizi del capitolo 3

Stile di codifica, tipi di dati ed array

Di seguito una serie di esercizi per fare pratica con quanto appreso nel capitolo 3. Ricordiamo che molti esercizi sono propedeutici ai successivi, quindi non conviene saltarne qualcuno o quantomeno vi consigliamo di consultare le soluzioni prima di andare avanti.

Esercizio 3.a)

Scrivere un semplice programma che svolga le seguenti operazioni aritmetiche correttamente, scegliendo accuratamente i tipi di dati da utilizzare per immagazzinare i risultati di esse.

1. Una divisione (usare il simbolo `/`) tra due interi `a = 5`, e `b = 3`. Immagazzinare il risultato in una variabile `r1`, scegliendone il tipo di dato opportunamente.
2. Una moltiplicazione (usare il simbolo `*`) tra un `char c = 'a'`, ed uno `short s = 5000`. Immagazzinare il risultato in una variabile `r2`, scegliendone il tipo di dato opportunamente.
3. Una somma (usare il simbolo `+`) tra un `int i = 6` ed un `float f = 3.14F`. Immagazzinare il risultato in una variabile `r3`, scegliendone il tipo di dato opportunamente.
4. Una sottrazione (usare il simbolo `-`) tra `r1`, `r2` e `r3`. Immagazzinare il risultato in una variabile `r4`, scegliendone il tipo di dato opportunamente.

Verificare la correttezza delle operazioni stampandone i risultati parziali e il risultato finale. Tenere presente la promozione automatica nelle espressioni, e utilizzare il casting opportunamente. Basta una classe con un `main()` che svolga le operazioni.

Esercizio 3.b)

Scrivere un programma con i seguenti requisiti.

- Implementare una classe `Persona` che dichiari le variabili `nome`, `cognome`, `eta` (età). Si dichiari inoltre un metodo `dettagli()` che restituisca in una stringa le informazioni sulla persona in questione. Ricordarsi di utilizzare le convenzioni e le regole descritte in questo capitolo.
- Implementare una classe `Principale` che, nel metodo `main()`, istanzi due oggetti chiamati `persona1` e `persona2` della classe `Persona`, inizializzando per ognuno di essi i relativi campi con sfruttamento dell'operatore dot.
- Dichiarare un terzo reference (`persona3`) che punti ad uno degli oggetti già istanziati. Controllare che effettivamente `persona3` punti all'oggetto voluto, stampando i campi di `persona3` sempre mediante l'operatore dot.
- Commentare adeguatamente le classi realizzate e sfruttare lo strumento javadoc per produrre la relativa documentazione.

Nella documentazione standard di Java sono usate tutte le regole e le convenzioni descritte in questo capitolo. Basta osservare che `String` inizia con lettera maiuscola, essendo una classe. Si può concludere che anche `System` è una classe.

Esercizio 3.c) Array, Vero o Falso:

1. Un array è un oggetto e quindi può essere dichiarato, istanziato ed inizializzato.
2. Un array bidimensionale è un array i cui elementi sono altri array.
3. Il metodo `length` restituisce il numero degli elementi di un array.
4. Un array non è ridimensionabile.
5. Un array è eterogeneo di default.

6. Un array di interi può contenere come elementi byte, ovvero le seguenti righe di codice non producono errori in compilazione:

```
int arr [] = new int[2];
byte a = 1, b=2;
arr [0] = a;arr [1] = b;
```

7. Un array di interi può contenere come elementi `char`, ovvero le seguenti righe di codice non producono errori in compilazione:

```
char a = 'a', b = 'b';
int arr [] = {a,b};
```

8. Un array di stringhe può contenere come elementi char, ovvero le seguenti righe di codice non producono errori in compilazione:

```
String arr [] = {'a' , 'b'};
```

9. Un array di stringhe è un array bidimensionale, perché le stringhe non sono altro che array di caratteri. Per esempio:

```
String arr [] = {"a" , "b"};
```

è un array bidimensionale.

10. Se abbiamo il seguente array bidimensionale:

```
int arr [][]= {
    {1, 2, 3},
    {1,2},
    {1,2,3,4,5}
};
```

risulterà che:

```
arr.length = 3;
arr[0].length = 3;
arr[1].length = 2;
arr[2].length = 5;
arr[0][0] = 1;
arr[0][1] = 2;
arr[0][2] = 3;
arr[1][0] = 1;
arr[1][1] = 2;
arr[1][2] = 3;
arr[2][0] = 1;
arr[2][1] = 2;
arr[2][2] = 3;
arr[2][3] = 4;
arr[2][4] = 5;
```

Esercizio 3.d)

Creare una classe `StampaMioNome` con un metodo `main()`, che stampa il vostro nome usando un array di caratteri.

Esercizio 3.e)

Creare una classe `Risultato` che dichiara una sola variabile d'istanza di tipo `float` di nome `risultato`. Aggiungere eventuali metodi e costruttori utili. Creare una classe `CambiaRisultato` che dichiara un metodo `public` di nome `cambiaRisultato()` che prende in input un oggetto di tipo `Risultato` e ne modifica la variabile interna `risultato` sommandola con un altro valore. Creare una classe con un metodo `main()` di nome `TestRisultato` che stampi la variabile `risultato` di un oggetto di tipo `Risultato`, prima e dopo che questo oggetto sia passato in input al metodo `cambiaRisultato()` di un oggetto di tipo `CambiaRisultato`.

Esercizio 3.f)

Dopo aver svolto l'esercizio precedente, aggiungere alla classe `CambiaRisultato` un metodo che si chiama sempre `cambiaRisultato()` che però prende in input una variabile `float` e ne modifica il risultato.
Creare poi una classe `TestRisultatoFloat` equivalente, che esegua le stesse operazioni della classe `TestRisultato` realizzata nell'esercizio precedente.

Esercizio 3.g)

Creare una classe `TestArgs` con un metodo `main()` che stampa la variabile `args[0]`. Quindi testarla passando vari input da riga di comando (cfr. paragrafo 3.6.5).

Esercizio 3.h)

La seguente classe dichiara vari identificatori di tipo stringa:

```
public class Esercizio3H {
    public String Break,
    String,
    character,
    bit,
    continues,
    exports,
    Class,
    imports,
```

```
    _AAA_;
    _@_,
    _;
}
```

Sono tutti validi? Una volta individuati quelli non validi, usare opportunamente i commenti per escluderli dalla compilazione.

Esercizio 3.i)

Data la seguente classe:

```
package com.claudiodesio.manuale.esercizi;

public class Esercizio3I {
    public final String nomeLinguaggio = "Java 9";
    public int intero;
    public void stampaStringa() {
        System.out.println(nomeLinguaggio);
    }
}
```

C'è qualche convenzione non rispettata per i nomi? Se si, quale e come modificare per risolvere? Se non si rispettano le convenzioni dei nomi, il codice non compila?

Esercizio 3.l)

Data la seguente classe:

```
public class Esercizio3L {
    public static void main(String args[]) {
        bit i1 = 8;
        short i2 = -1024;
        integer i3 = 638;
        long i5 = 888_666_777;
        float i6 = 0;
        double i7 = 0x11B;
    }
}
```

Le variabili all'interno del metodo `main()` sono tutte dichiarate correttamente? E i valori assegnati sono tutti entro l'intervallo di rappresentazione dei rispettivi tipi?

Esercizio 3.m)

Data la seguente classe:

```
public class Esercizio3M {
    public static void main(String args[]) {
```

```
        boolean b = true;
        char c = 'I';
        System.out.println(b);
        System.out.println(c+1);
    }
}
```

Tenendo presente che la lettera `I`, è codificata tramite il numero 73, quale dei seguenti output produrrà una volta eseguita?

1. Nessuno, avremo un errore in compilazione.
2. `true` alla prima riga e `74` alla seconda.
3. `true` alla prima riga e `L` alla seconda.
4. `true` alla prima riga e `J` alla seconda.
5. `0` alla prima riga e `74` alla seconda.
6. `0` alla prima riga e `J` alla seconda.
7. `0` alla prima riga e `L` alla seconda.

Esercizio 3.n)

Data la seguente classe:

```
public class Esercizio3N {
    public static void main(String args[]) {
        String s = "Jav";
        char c = "a";
        System.out.println(s+c+1);
    }
}
```

quale dei seguenti output produrrà una volta eseguita?

1. Nessuno, avremo un errore in compilazione.
2. `Java`
3. `Java1`
4. `Javb`

Esercizio 3.o)

Data la classe:

```
package parcheggio;

public class Auto {
    public String tipo;

    public Auto(String t) {
        tipo = t;
    }
}
```

Affinché la seguente classe compili:

```
package lavoratori;
//Inserisci il codice qui

public class Autista {
    public void guida(Auto auto) {
        System.out.println("Sto guidando l'auto " + auto.tipo);
    }
}
```

Bisogna inserire una riga di codice. Quale (o quali, potrebbero essere valide anche più di una) tra le seguenti righe permetterebbe alla classe `Autista` di essere compilata?

1. `import parcheggio.Auto;`
2. `import parcheggio.*;`
3. `import parcheggio. lavoratori.*;`
4. `import parcheggio.Auto.*;`
5. `import parcheggio.*.Auto;`
6. `import lavoratori.parcheggio.Auto;`

Esercizio 3.p)

Considerando le classi `Auto` e `Autista` dell'esercizio precedente, che pezzo di codice bisogna aggiungere alla seguente classe:

```
public class Esercizio3P {
//Inserisci il codice qui

    public static void main(String args[]) {
        Auto auto = new Auto("Toyota Yaris");
        Autista autista = new Autista();
        autista.guida(auto);
    }
}
```

Scegliere una delle seguenti opzioni:

1. `import parcheggio.*;`
2. `import lavoratori.*;`
3. `import parcheggio.Auto;` e `import lavoratori.Autista;`
4. Non bisogna aggiungere codice. Il codice è già compilabile.

Esercizio 3.q)

La seguente affermazione è corretta?

- ❑ Quando passiamo un reference di un oggetto in input ad un metodo, siamo sicuri che terminata l'esecuzione del metodo, il nostro reference continuerà a puntare sempre allo stesso oggetto a cui puntava prima dell'esecuzione del metodo. Questo non significa che la struttura interna dell'oggetto, non possa essere modificata all'interno del metodo. Infatti il parametro locale del metodo, assumerà lo stesso indirizzamento del reference passato, e potrà quindi lavorare sullo stesso oggetto.

Esercizio 3.r)

Scrivere un programma che prenda in input un argomento (variabile `args` del metodo `main()`) e lo immagazzini come terzo elemento di un array di stringhe locale chiamato per l'appunto `array`.

Questo programma funzionerà solo se viene passato almeno un argomento. Eventuali parametri passati da riga di comando oltre al primo, saranno ignorati dal programma.

Esercizio 3.s)

Data la classe:

```
public class Auto {
    public String tipo;

    public Auto(String t) {
        tipo = t;
    }
}
```

Creare una classe `Nave` che astragga il concetto di nave che carica automobili. Questa classe deve definire un metodo `caricaAuto()` a cui sarà passato un oggetto `Auto`. Ogni oggetto auto verrà immagazzinato in una variabile d'istanza array chiamata `autoArray`, nel primo posto disponibile dell'array. Creare anche una classe di test che è possibile chiamare `Esercizio3S`.

Esercizio 3.t)

Creare un programma `Esercizio3T` che deve essere eseguito passando un parametro da riga di comando rappresentante un numero intero, nel seguente modo:

```
java Esercizio3T 9
```

Con EJE è possibile passare parametri con la scorciatoia di tastiera Maiuscolo - F9, ovvero con il menu esegui con argomenti (execute with args).

Si può sostituire il numero intero `9` con un qualsiasi altro numero intero.
Il programma dovrà:

1. utilizzare `args[0]`, che contiene un numero intero, per creare un array di interi;
2. stampare una frase che confermerà la creazione dell'array, stampandone la dimensione.

Visto che i parametri vengono immagazzinati negli elementi dell'array di stringhe `args`, bisogna convertire il parametro passato in input da stringa ad intero. Cercare nella libreria il metodo `parseInt` della classe `Integer`. Leggere la documentazione, capire come funziona ed utilizzarlo nel programma.

Nel caso ci si trovi in difficoltà si esegua una ricerca su Google per cercare aiuto, il web è pieno di soluzioni.

Esercizio 3.u)

Creare un programma che:

1. crei un array di tipi carattere, con tutte le lettere dell'alfabeto;
2. utilizzi un metodo (statico) della classe `java.util.Arrays` per stampare il suo contenuto sotto forma di stringa. Cercare nella documentazione ufficiale il metodo adatto allo scopo.

Nel caso ci si trovi in difficoltà si esegua una ricerca su Google per cercare aiuto, il web è pieno di soluzioni.

Esercizio 3.v)

Creare una classe che stampi numeri casuali di tipo intero.

Suggerimento: esiste un metodo di una classe nella libreria Java che fa esattamente questo. Per trovarlo, si parta dal tradurre in inglese la parola "casuale" e si cerchi nella documentazione.

Esercizio 3.z)

Creare una classe `Pagella` che astragga il concetto di pagella scolastica. Essa deve avere le seguenti informazioni:

1. nome, cognome e classe dell'alunno;
2. una tabella che associ per ogni materia, il voto e il giudizio.
3. Deve inoltre dichiarare un metodo che stampi in maniera leggibile i dati della pagella.

Creare anche una classe `Esercizio3Z` che stampi una o più pagelle.

Soluzioni degli esercizi del capitolo 3

Soluzione 3.a)

```
public class Esercizio3A {
    public static void main(String args[]) {
        int a = 5, b = 3;
        double r1 = (double)a/b;
        System.out.println("r1 = " + r1);
        char c = 'a';
        short s = 5000;
        int r2 = c*s;
        System.out.println("r2 = " + r2);
        int i = 6;
        float f = 3.14F;
        float r3 = i + f;
        System.out.println("r3 = " + r3);
        double r4 = r1 - r2 - r3;
        System.out.println("r4 = " + r4);
    }
}
```

Soluzione 3.b)

```
public class Persona {
    public String nome;
    public String cognome;
    public int eta;
    public String dettagli() {
        return nome + " " + cognome + " anni " + eta;
```

```
    }
}
public class Principale {
    public static void main(String args []) {
        Persona persona1 = new Persona();
        Persona persona2 = new Persona();
        persona1.nome = "Mario";
        persona1.cognome = "Rossi";
        persona1.eta = 30;
        System.out.println("persona1 "+persona1.dettagli());
        persona2.nome = "Giuseppe";
        persona2.cognome = "Verdi";
        persona2.eta = 40;
        System.out.println("persona2 "+persona2.dettagli());
        Persona persona3 = persona1;
        System.out.println("persona3 "+persona3.dettagli());
    }
}
```

Soluzione 3.c) Array, Vero o Falso:

1. **Vero**.
2. **Vero**.
3. **Falso**, la variabile `length` restituisce il numero degli elementi di un array.
4. **Vero**.
5. **Falso**.
6. **Vero**, un `byte` (che occupa solo 8 bit) può essere immagazzinato in una variabile `int` (che occupa 32 bit).
7. **Vero**, un `char` (che occupa 16 bit) può essere immagazzinato in una variabile `int` (che occupa 32 bit).
8. **Falso**, un `char` è un tipo di dato primitivo e `String` è una classe. I due tipi di dati non sono compatibili.
9. **Falso**, in Java la stringa è un oggetto istanziato dalla classe `String` e non un array di caratteri.
10. **Falso**, tutte le affermazioni sono giuste tranne `arr[1][2] = 3;` perché questo elemento non esiste.

Soluzione 3.d)

Il listato dovrebbe essere simile al seguente:

```
public class StampaMioNome {
    public static void main(String args[]) {
        char [] nome = {'C', 'l', 'a', 'u', 'd', 'i', 'o'};
        System.out.println(nome);
    }
}
```

Soluzione 3.e)

Il listato della classe `Risultato` potrebbe essere il seguente:

```
public class Risultato {
    public float risultato;

    public Risultato(float ris) {
        risultato = ris;
    }

    public void stampa() {
        System.out.println(risultato);
    }
}
```

Si noti che abbiamo creato un costruttore e un metodo di stampa di comodità.
Il listato della classe `CambiaRisultato` potrebbe essere il seguente:

```
public class CambiaRisultato {
    public void cambiaRisultato(Risultato risultato) {
        risultato.risultato += 1;
    }
}
```

Il listato della classe `TestRisultato` potrebbe essere il seguente:

```
public class TestRisultato {
    public static void main(String args[]) {
        Risultato risultato = new Risultato(5.0F);
        risultato.stampa();
        CambiaRisultato cr = new CambiaRisultato();
        cr.cambiaRisultato(risultato);
        risultato.stampa();
    }
}
```

L'output del precedente codice sarà:

```
5.0
6.0
```

Soluzione 3.f)

Il listato della classe `CambiaRisultato` dovrebbe cambiare come segue:

```
public class CambiaRisultato {
    public void cambiaRisultato(Risultato risultato) {
        risultato.risultato += 1;
    }

    public float cambiaRisultato(float risultato) {
        risultato += 1;
        return risultato;
    }
}
```

Si noti che questa volta il metodo deve ritornare il nuovo valore della variabile trattandosi di una variabile di tipo primitivo (cfr. paragrafo 3.3).
Il listato della classe `TestRisultatoFloat` potrebbe essere il seguente:

```
public class TestRisultatoFloat {
    public static void main(String args[]) {
        float risultato = 5.0F;
        System.out.println(risultato);
        CambiaRisultato cr = new CambiaRisultato();
        risultato = cr.cambiaRisultato(risultato);
        System.out.println(risultato);
    }
}
```

Si noti che abbiamo dovuto riassegnare il valore della variabile `risultato` dopo la computazione del metodo `cambiaRisultato()`.

Soluzione 3.g)

Il listato dovrebbe essere simile a:

```
public class TestArgs {
    public static void main(String args[]) {
        System.out.println(args[0]);
    }
}
```

Si noti che se non passate un argomento quando eseguite l'applicazione otterrete un'eccezione al runtime:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
      at TestArgs.main(TestArgs.java:3)
```

Le eccezioni sono trattate nel capitolo 9.

Soluzione 3.h)

Solo gli ultimi due identificatori non sono validi, infatti:

1. L'identificatore `Break` è differente da `break` (tutte le parole chiave non hanno lettere maiuscole).
2. L'identificatore `String` coincide con il nome della classe `String`, ma non essendo una parola chiave, è possibile utilizzarla come identificatore. Ciononostante si tratta di una cattiva pratica.
3. L'identificatore `character` non è una parola chiave (lo è invece `char`).
4. L'identificatore `bit` non è una parola chiave (lo è invece `byte`).
5. L'identificatore `continues` non è una parola chiave (lo è invece `continue`).
6. L'identificatore `exports` è una parola a utilizzo limitato, e sarebbe inutilizzabile all'interno della dichiarazione di un modulo, ma in questo contesto non crea problemi.
7. L'identificatore `Class` non è una parola chiave (lo è invece `class`, vedi punto 1).
8. L'identificatore `imports` non è una parola chiave (lo è invece `import`).
9. L'identificatore `_AAA_` non è una parola chiave.
10. L'identificatore `_@_` non è legale perché no si può usare il simbolo di chiocciola negli identificatori.
11. L'identificatore `_` è una parola riservata a partire da Java 9.

È possibile commentare i due identificatori non validi nel seguente modo:

```
public class Esercizio3H {
    public String Break,
    String,
    character,
    bit,
    continues,
    exports,
    Class,
    imports,
    _AAA_/*,
    _@_,
    _*/;
}
```

Utilizzando un commento su più righe per commentare solo ciò che va commen-

tato. Chiaramente questo approccio va a discapito della leggibilità del codice. Il modo più opportuno per commentare richiede la cancellazione del simbolo di "`,`" con il simbolo "`;`" subito dopo la dichiarazione dell'identificatore `_AAA_`, e l'utilizzo del commento a singola riga:

```
public class Esercizio3H {
    public String Break,
    String,
    character,
    bit,
    continues,
    exports,
    Class,
    imports,
    _AAA_;
//    _@_,
//    _;
}
```

Soluzione 3.i)

L'unica convenzione non utilizzata correttamente è quella per la costante (si ricorda che le convenzioni non influiscono sulla compilabilità del codice).
Il codice andrebbe corretto nel seguente modo:

```
package com.claudiodesio.manuale.esercizi;

public class Esercizio3ISoluzione {
    public final String NOME_LINGUAGGIO = "Java 9";
    public int intero;
    public void stampaStringa() {
        System.out.println(NOME_LINGUAGGIO);
    }
}
```

Soluzione 3.l)

I tipi `bit` e `integer` non esistono (semmai esistono `byte` e `int`). Tutti i valori dichiarati rientrano nei rispettivi intervalli di rappresentazione, compreso il valore `0x11B` che vale `283.0` per il sistema decimale e che, essendo immagazzinato in un `double`, è assolutamente compatibile.

Soluzione 3.m)

La risposta giusta è la numero 2. Infatti un literal booleano stampa esattamente il suo valore literal `true`. Invece `c+1` viene promosso a numero intero, e da `73` diven-

ta 74. Per potergli far stampare il valore carattere relativo (J) bisognerebbe fare un cast su tutta l'operazione in questo modo:

```
        System.out.println((char)(c+1));
```

Soluzione 3.n)

Avremo un errore in compilazione perché il valore assegnato al carattere c, è una stringa (si notino le virgolette al posto degli apici).

Soluzione 3.o)

Le risposte corrette sono la 1 e la 2, tutte le altre sono scorrette a livello sintattico.

Soluzione 3.p)

La risposta corretta è la 3 perché sono state utilizzate nel codice entrambe le classi `Auto` e `Autista`.

Soluzione 3.q)

Sì, il ragionamento è corretto.

Soluzione 3.r)

Il listato dovrebbe essere simile a:

```
public class EsercizioR {
    public static void main(String args[]) {
        String[] array = new String[5];
        array[2] = args[0];
    }
}
```

Soluzione 3.s)

Il codice richiesto dovrebbe essere simile al seguente:

```
public class Nave {

    int indice = 0;
    public Auto[] autoArray;

    public Nave() {
        autoArray = new Auto[100];
    }
```

```
    public void caricaAuto(Auto auto) {
        autoArray[indice] = auto;
        System.out.println("Auto: " + auto.tipo + " caricata");
        indice++;
    }
}
```

Dove abbiamo usato un indice per tenere traccia delle posizioni già occupate nella nave. Questo viene incrementato ogni volta che viene caricata un'auto, e poi sfruttato per caricare la prossima.
La seguente classe di test soddisfa la richiesta:

```
public class Esercizio3S {
    public static void main(String args[]) {
        Nave nave = new Nave();
        Auto auto1 = new Auto("Renault");
        Auto auto2 = new Auto("Volkswagen");
        Auto auto3 = new Auto("Nissan");
        nave.caricaAuto(auto1);
        nave.caricaAuto(auto2);
        nave.caricaAuto(auto3);
    }
}
```

Soluzione 3.t)

Il listato richiesto dovrebbe essere simile al seguente:

```
public class Esercizio3T {
    public static void main(String args[]) {
        int dimensioneArray = Integer.parseInt(args[0]);
        int [] array = new int[dimensioneArray];
        System.out.println("L'array ha dimensione " + array.length);
    }
}
```

Si noti che il metodo `parseInt()` è statico (argomento non ancora affrontato) e può essere utilizzato con la sintassi: `nomeClasse.parseInt()`, ma è possibile anche istanziare un oggetto e invocarlo come un normale metodo (ma è inutile istanziare l'oggetto).

Soluzione 3.u)

Il listato richiesto dovrebbe essere simile al seguente:

```
import java.util.Arrays;

public class Esercizio3U {
```

```
    public static void main(String args[]) {
        char[] array = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
          'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'z'};
        System.out.println(Arrays.toString(array));
    }
}
```

Il metodo `toString()` della classe `Arrays` era il metodo richiesto. Per utilizzarlo è necessario importare la classe `Arrays`.

Soluzione 3.v)

Il listato richiesto dovrebbe essere simile al seguente:

```
import java.util.Random;

public class Esercizio3V {
    public static void main(String args[]) {
        Random random = new Random();
        System.out.println(random.nextInt());
    }
}
```

Soluzione 3.z)

Decidiamo di creare un'astrazione per la classe `Studente`:

```
public class Studente {
    public String nome;
    public String cognome;
    public String classe;

    public Studente(String nom, String cog, String cla) {
        nome = nom;
        cognome = cog;
        classe = cla;
    }

    public String toString() {
        return "Studente: " + nome + " " + cognome + "\nClasse " + classe;
    }
}
```

Abbiamo dichiarato le informazioni essenziali (richieste dalla traccia dell'esercizio) e abbiamo creato un costruttore per impostare queste informazioni. Abbiamo anche creato un metodo `toString()` che restituisce una stringa descrittiva dell'oggetto.

Vedremo più avanti che questo metodo sarà utilizzato spessissimo nella programmazione Java, perché già presente in ogni classe.

Poi creiamo una classe `Pagella` che astrae il concetto di tabella:

```
import java.util.Arrays;

public class Pagella {
    public Studente studente;
    public String[][] tabellaVoti;

    public Pagella(Studente stu, String [][] tab) {
        studente = stu;
        tabellaVoti = tab;
    }

    public void stampaPagella() {
        System.out.println(studente.toString());
        System.out.println(Arrays.toString(tabellaVoti[0]));
        System.out.println(Arrays.toString(tabellaVoti[1]));
        System.out.println(Arrays.toString(tabellaVoti[2]));
        System.out.println(Arrays.toString(tabellaVoti[3]));
        System.out.println(Arrays.toString(tabellaVoti[4]));
        System.out.println(Arrays.toString(tabellaVoti[5]));
        System.out.println(Arrays.toString(tabellaVoti[6]));
    }
}
```

Si noti che questa classe dichiara un oggetto `studente` e un array bidimensionale `tabellaVoti`, entrambi da impostare quando si istanzia l'oggetto con il costruttore fornito. Inoltre dichiara il metodo `stampaPagella()` che usa il metodo statico `toString()` della classe `java.util.Arrays` per formattare il contenuto di ogni "riga" dell'array bidimensionale `tabellaVoti`.
Infine con la seguente classe, andiamo a stampare due pagelle:

```
public class Esercizio3Z {
    public static void main(String args[]) {
        Studente studente1 = new Studente("Giovanni","Battista","5A");
        String [][] tabellaVoti1 = {
            {"Italiano","7","Non si impegna troppo."},
            {"Matematica","9","È molto portato per questa materia."},
            {"Storia","7","Potrebbe fare di più."},
            {"Geografia","8","Appassionato."},
            {"Inglese","9", "Capace di sostenere dialoghi."},
```

```
            {"Scienze Motorie","6", "Voto d'incoraggiamento."},
            {"Musica","7", "Ha una certa passione per la materia."}
        };
        Pagella pagella1 = new Pagella(studente1, tabellaVoti1);

        Studente studente2 = new Studente("Daniele","Sapore","2A");
        String [][] tabellaVoti2 = {
            {"Italiano","8","Manifesta entusiasmo per la materia."},
            {"Matematica","5","Per niente interessato."},
            {"Storia","6","Interessato, ma si impegna poco."},
            {"Geografia","6","Potrebbe fare di più."},
            {"Inglese","8", "Ottima pronuncia."},
            {"Scienze Motorie","7", "Un po' pigro."},
            {"Musica","9", "Suona diversi strumenti ed ha un'ottima voce."}
        };
        Pagella pagella2 = new Pagella(studente2, tabellaVoti2);

        pagella1.stampaPagella();
        pagella2.stampaPagella();
    }
}
```

L'output è il seguente:

```
Studente: Giovanni Battista
Classe 5A
[Italiano, 7, Non si impegna troppo.]
[Matematica, 9, È molto portato per questa materia.]
[Storia, 7, Potrebbe fare di più.]
[Geografia, 8, Appassionato.]
[Inglese, 9, Capace di sostenere dialoghi.]
[Scienze Motorie, 6, Voto d'incoraggiamento.]
[Musica, 7, Ha una certa passione per la materia.]
Studente: Daniele Sapore
Classe 2A
[Italiano, 8, Manifesta enstusiamo per la materia.]
[Matematica, 5, Per niente interessato.]
[Storia, 6, Interessato, ma si impegna poco.]
[Geografia, 6, Potrebbe fare di più.]
[Inglese, 8, Ottima pronuncia.]
[Scienze Motorie, 7, Un po' pigro.]
[Musica, 9, Suona diversi strumenti ed ha un'ottima voce.]
```

Esercizi del capitolo 4

Operatori e Gestione del flusso di esecuzione

Dopo aver studiato il capitolo 4 dovremmo già essere in grado di poter scrivere programmi usando il linguaggio Java. Quello che manca ancora sono le nozioni di Object Orientation che si studieranno per bene a partire dal prossimo capitolo. Intanto prendiamo dimestichezza con il flusso di esecuzione.

Esercizio 4.a)

Scrivere un semplice programma, costituito da un'unica classe, che sfruttando esclusivamente un ciclo infinito, l'operatore modulo, due costrutti `if`, un `break` ed un `continue`, stampi solo i primi cinque numeri pari.

Esercizio 4.b)

Scrivere un'applicazione che stampi i 26 caratteri dell'alfabeto (inglese) con un ciclo.

Esercizio 4.c)

Scrivere una semplice classe che stampi a video la tavola pitagorica.

Suggerimento 1: non sono necessari array.

Suggerimento 2: il metodo `System.out.println()` stampa l'argomento che gli viene passato e poi sposta il cursore alla riga successiva; infatti `println` sta per "print line". Esiste anche il metodo `System.out.print()` che invece stampa solamente il parametro passatogli.

Suggerimento 3: sfruttare un doppio ciclo innestato.

Esercizio 4.d) Operatori e flusso di esecuzione, Vero o Falso:

1. Gli operatori unari di pre-incremento e post-incremento applicati ad una variabile danno lo stesso risultato, ovvero se abbiamo:

```
int i = 5;
```

sia

```
i++;
```

sia

```
++i;
```

aggiornano il valore di `i` a `6`;

2. `d += 1` è equivalente a `d++` dove `d` è una variabile `double`.

3. Se abbiamo:

```
int i = 5;
int j = ++i;
int k = j++;
int h = k--;
boolean flag = ((i != j) && ( (j <= k ) ||  (i <= h)) );
```

`flag` avrà valore `false`.

4. L'istruzione:

```
System.out.println(1 + 2 + "3");
```

stamperà `33`.

5. Il costrutto `switch` può in ogni caso sostituire il costrutto `if`.

6. L'operatore ternario può in ogni caso sostituire il costrutto `if`.

7. Il costrutto `for` può in ogni caso sostituire il costrutto `while`.
8. Il costrutto `do` può in ogni caso sostituire il costrutto `while`.
9. Il costrutto `switch` può in ogni caso sostituire il costrutto `while`.
10. I comandi `break` e `continue` possono essere utilizzati nei costrutti `switch`, `for`, `while` e `do` ma non nel costrutto `if`.

Esercizio 4.e)

Modificare la classe `TestArgs` creata nell'esercizio 3.g, in modo tale da evitare eccezioni al runtime, con un costrutto imparato in questo capitolo.

Esercizio 4.f)

È buona norma aggiungere la clausola `default` ad un costrutto `switch`. Sapreste spiegare perché?

Esercizio 4.g)

È buona norma aggiungere la clausola `else` ad un costrutto `if`. Sapreste spiegare perché?

Esercizio 4.h)

Creare una classe con un metodo `main()` che selezioni i primi 10 numeri divisibili per 3, e li stampi dopo averli concatenati con una stringa in modo tale che l'output del programma sia:

```
Numero multiplo di 3 = 3
Numero multiplo di 3 = 6
Numero multiplo di 3 = 9
...
```

Usare un ciclo `for` ordinario.

Esercizio 4.i)

Ripetere l'esercizio 4.h usando un ciclo `while` invece del ciclo `for`.

Esercizio 4.l)

Ripetere l'esercizio 4.h usando un ciclo `do-while` invece del ciclo `for`.

Esercizio 4.m)

Si consideri il seguente codice:

```
import java.util.Scanner;

public class ProgrammaInterattivo {
    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        String stringa = "";
        System.out.println("Digita qualcosa e batti enter, oppure scrivi "
          + \"fine\" per terminare il programma");
        while(!(stringa = scanner.next()).equals("fine")) {
            System.out.println("Hai digitato " + stringa.toUpperCase() + "!");
        }
        System.out.println("Fine programma!");
    }
}
```

Questa classe legge l'input da tastiera mediante la classe `Scanner` del package `java.util` (di cui parleremo solo nel capitolo 14). Il metodo `next()` usato nel costrutto `while` (con una sintassi complessa che comprende anche l'assegnazione alla variabile `stringa`) è un metodo bloccante (ovvero che blocca l'esecuzione del codice in attesa di input da parte dell'utente) che legge l'input da tastiera sino a quando si preme il tasto enter (invio). Il programma termina quando si digita la parola "fine".

Attenzione che su EJE i metodi bloccanti di `Scanner` leggono le pressioni dei tasti. Questo significa che se vogliamo digitare una parola in maiuscolo, siamo costretti a fare clic sul tasto maiuscolo, che verrà evidenziato dall'output con un simbolo di "?".

Modificare il programma precedente in modo tale che diventi un moderatore di parole, ovvero che censuri alcune parole digitate.

Eseguire l'esercizio solo censurando le parole digitate singolarmente (non all'interno di una frase...), a meno che non si è convinti di essere in grado di farlo (eventualmente la documentazione è come sempre a vostra disposizione per cercare metodi utili alla causa).

Esercizio 4.n)

Quali dei seguenti operatori può essere utilizzato con variabili booleane?

1. `+`
2. `%`
3. `++`
4. `/=`
5. `&`
6. `=`
7. `!!`
8. `>>>`

Esercizio 4.o)

Qual è l'output del seguente programma?

```
public class Esercizio4O {
    public static void main(String args[]) {
        int i = 99;
        if (i++ >= 100) {
            System.out.println(i+=10);
        } else {
            System.out.println(--i==99?i++:++i);
        }
    }
}
```

Scegliere tra le seguenti opzioni:

1. `10`
2. `101`
3. `99`
4. `110`

Esercizio 4.p)

Qual è l'output del seguente programma?

```
public class Esercizio4P {
    public static void main(String args[]) {
        int i = 22;
```

```
        int j = i++%3;
        i = j!=0?j:i;
        switch (i) {
            case 1:
            System.out.println(8<<2);
            case 0:
            System.out.println(8>>2);
            break;
            case 2:
            System.out.println(i!=j);
            break;
            case 3:
            System.out.println(++j);
            break;
            default:
            System.out.println(i++);
            break;
        }
    }
}
```

Scegliere tra le seguenti opzioni:

1. `24`
2. `6` e al rigo successivo `10`
3. `10`
4. `true`
5. `false`
6. `22`
7. `21`
8. `32` e al rigo successivo `2`

Esercizio 4.q)

Si scriva un programma che chieda all'utente di inserire il numero di giorni passati dall'ultima vacanza fatta. Una volta inserito questo numero, il programma dovrà rispondere quanti minuti sono passati dall'ultima vacanza.

Esercizio 4.r)

Data la seguente classe:

```
public class Esercizio4R {
```

```
    private static int matrice[][] = {
        {1, 7, 3, 9, 5, 3},
        {6, 2, 3},
        {7, 5, 1, 4, 0},
        {1, 0, 2, 9, 6, 3, 7, 8, 4}
    };

    public static void main(String args[]) {

    }
}
```

implementare il metodo `main()` in modo che legga un numero (tra `0` e `9`) come parametro `args[0]`, e trovi la posizione (riga e colonna) della prima occorrenza del numero specificato all'interno dell'array bidimensionale denominato `matrice`.

Esercizio 4.s)

La soluzione dell'esercizio precedente fallisce nel momento in cui:

1. non si specifica un argomento da riga di comando;
2. si specifica da riga di comando un argomento intero che non rientra nel range 0-9;
3. si specifica da riga di comando un argomento che non è un numero intero;

Aggiungere, alla soluzione dell'esercizio 4.r, il codice che gestisce i tre casi (specificando un messaggio con le istruzioni da seguire per il corretto utilizzo).

Il terzo caso può essere gestito con quanto studiato finora, ma più avanti studieremo dei metodi per semplificare il nostro codice. In particolare nel capitolo 9 dove si parla di gestione delle eccezioni, e nel capitolo 14 in cui parleremo delle regular expression, vedremo che esistono delle soluzioni piuttosto semplici per gestire il terzo caso.

Esercizio 4.t)

Dichiarare una classe `CalcolatriceSemplificata` che dati due numeri definisca i metodi per:

1. Sommarli.
2. Sottrarre il secondo dal primo.
3. Moltiplicarli.
4. Dividerli.
5. Restituire il resto della divisione.
6. Restituire il numero più grande (il massimo).
7. Restituire il numero più piccolo (il minimo).
8. Restituire la media dei due numeri.
9. Creare una classe che testa il funzionamento di tutti i metodi.

Esercizio 4.u)

Dichiarare una classe che utilizza la classe `Scanner`, che permette all'utente di interagire con la classe `CalcolatriceSemplificata`: l'utente deve poter scrivere il primo operando, selezionare da una lista l'operazione da eseguire e specificare il secondo operando. Il programma deve restituire il giusto risultato.

Attenzione che i metodi bloccanti di `Scanner` leggono le pressioni dei tasti. Questo significa che se vogliamo digitare una parola in maiuscolo, siamo costretti a fare clic sul tasto maiuscolo, che verrà evidenziato dall'output con un simbolo di "?".

Esercizio 4.v)

Dichiarare una classe `StranaCalcolatrice` che dati un numero imprecisato di numeri definisca i metodi per:

1. Sommarli.
2. Sottrarre il secondo dal primo.
3. Moltiplicarli.
4. Dividerli.

5. Restituire il resto della divisione.
6. Restituire il numero più grande (il massimo).
7. Restituire il numero più piccolo (il minimo).
8. Restituire la media dei due numeri.

Esercizio 4.z)

Dichiarare una classe che utilizza la classe `Scanner`, che permette all'utente di interagire con la classe `StranaCalcolatrice`. Al lettore è lasciata la decisione di come l'utente dovrà interagire con il programma.

> **Attenzione che i metodi bloccanti di `Scanner` leggono le pressioni dei tasti. Questo significa che se vogliamo digitare una parola in maiuscolo, siamo costretti a fare clic sul tasto maiuscolo, che verrà evidenziato dall'output con un simbolo di "?".**

Soluzioni degli esercizi del capitolo 4

Soluzione 4.a)

```
public class TestPari {
    public static void main(String args[]) {
        int i = 0;
        while (true) {
            i++;
            if (i > 10)
                break;
            if ((i % 2) != 0)
                continue;
            System.out.println(i);
        }
    }
}
```

Nel metodo `main()` dichiariamo prima una variabile `i` che fa da indice e che inizializziamo a `0`. Poi dichiariamo il ciclo infinito `while`, la cui condizione è sempre `true`. All'interno del ciclo incrementiamo subito di un'unità il valore della variabile `i`. Poi controlliamo se il valore della suddetta variabile è maggiore di `10`, se la risposta è sì, il costrutto `break` seguente ci farà uscire dal ciclo che doveva essere infinito, e di conseguenza il programma terminerà subito dopo. Nel caso invece `i` valesse meno di `10`, utilizzando l'operatore modulo (`%`) si controlla se il resto della divisione tra `i` e `2` è diverso da `0`. Ma tale resto sarà diverso da `0` se e solo se `i` è un numero dispari. Se quindi il numero è dispari, con il costrutto `continue` passeremo alla prossima iterazione partendo dalla prima istruzione del ciclo `while` (`i++`).

Se invece `i` è un numero pari, allora esso verrà stampato.
L'output del precedente programma è il seguente:

```
2
4
6
8
10
```

Soluzione 4.b)

```
public class TestArray {
    public static void main(String args[]) {
        for (int i = 0; i < 26; ++i) {
            char c = (char)('a' + i);
            System.out.println(c);
        }
    }
}
```

Si esegue un ciclo con l'indice `i` che varia da `0` a `25`. Aggiungendo al carattere `'a'` il valore dell'indice `i` (che ad ogni iterazione si incrementa di un'unità), otterremo gli altri caratteri dell'alfabeto. Il cast a `char` è necessario perché la somma tra un carattere ed un intero viene promosso ad intero.

Soluzione 4.c)

Il listato potrebbe essere il seguente:

```
public class Tabelline {
    public static void main(String args[]) {
        for (int i = 1; i <= 10; ++i) {
            for (int j = 1; j <= 10; ++j) {
                System.out.print(i*j + "\t");
            }
            System.out.println();
        }
    }
}
```

Con questo doppio ciclo innestato ed utilizzando il carattere di escape di tabulazione (`\n`), riusciamo a stampare la tavola pitagorica con poche righe di codice.

Soluzione 4.d) Operatori e flusso di esecuzione, Vero o Falso:

1. **Vero**.

2. **Vero**.
3. **Falso**, la variabile booleana `flag` avrà valore `true`. Le espressioni *atomiche* valgono rispettivamente `true-false-true`, sussistendo le seguenti uguaglianze: `i = 6`, `j = 7`, `k = 5`, `h = 6`. Infatti `(i != j)` vale `true` e inoltre `(i <= h)` vale `true`. L' espressione `( (j <= k ) || (i <= h)) )` vale `true`, sussistendo l'operatore OR. Infine l'operatore AND fa sì che la variabile `flag` valga `true`.
4. **Vero**.
5. **Falso**, `switch` può testare solo una variabile intera (o compatibile) confrontandone l'uguaglianza con costanti (in realtà dalla versione 5 si possono utilizzare come variabili di test anche le enumerazioni e il tipo `Integer`, e dalla versione 7 anche le stringhe). Il costrutto `if` permette di svolgere controlli incrociati sfruttando differenze, operatori booleani etc.
6. **Falso**, l'operatore ternario è sempre vincolato ad un'assegnazione del risultato ad una variabile. Questo significa che produce sempre un valore da assegnare e da utilizzare in qualche modo (per esempio passando un argomento invocando un metodo). Per esempio, se `i` e `j` sono due interi, la seguente espressione: `i < j ? i : j;` provocherebbe un errore in compilazione (oltre a non avere senso).
7. **Vero**.
8. **Falso**, il `do` in qualsiasi caso garantisce l'esecuzione della prima iterazione sul codice. Il `while` potrebbe prescindere da questa soluzione.
9. **Falso**, lo `switch` è una condizione non un ciclo.
10. **Falso**, il `continue` non si può utilizzare nello `switch` ma solo nei cicli.

Soluzione 4.e)

Il listato potrebbe essere il seguente:

```
public class TestArgs {
    public static void main(String args[]) {
        if (args.length == 1) {
            System.out.println(args[0]);
        } else {
            System.out.println("Specificare un valore da riga di comando");
        }
    }
}
```

Soluzione 4.f)

Questo perché non sappiamo a priori come si evolverà il nostro programma, e quindi, anche se nel momento in cui si scrive il programma potrebbe non essere necessario la clausola `default`, modificando il programma potrebbe nascere una nuova condizione non prevista, creando un baco nella nostra applicazione. Infatti il nuovo caso non sarà contemplato e il flusso di esecuzione non entrerà in nessuna clausola del costrutto `switch`. Anche usare il costrutto `default` solo per stampare una frase "Caso non previsto" potrebbe essere una buona abitudine.

Soluzione 4.g)

La risposta è identica alla precedente. La clausola `else` per un `if` è equivalente a una clausola `default` per lo `switch`.

Soluzione 4.h)

Il listato richiesto potrebbe essere il seguente:

```
public class Esercizio4H {
    public static void main(String args[]) {
        for (int i = 1, j = 1; j <= 10; i++) {
            if (i % 3 == 0) {
                System.out.println("Numero multiplo di 3 = " + i);
                j++;
            }
        }
    }
}
```

Soluzione 4.i)

Il listato richiesto potrebbe essere il seguente:

```
public class Esercizio4i {
    public static void main(String args[]) {
        int i = 1, j = 1;
        while (j <= 10) {
            if (i % 3 == 0) {
                System.out.println("Numero multiplo di 3 = " + i);
                j++;
            }
            i++;
        }
    }
}
```

Soluzione 4.l)

Il listato richiesto potrebbe essere il seguente:

```
public class Esercizio4l {
    public static void main(String args[]) {
        int i = 1, j = 1;
        do {
            if (i % 3 == 0) {
                System.out.println("Numero multiplo di 3 = " + i);
                j++;
            }
            i++;
        } while(j <= 10);
    }
}
```

Soluzione 4.m)

Il listato richiesto potrebbe essere il seguente:

```
import java.util.Scanner;

public class Moderatore {

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        String stringa = "";
        System.out.println("Digita qualcosa e batti enter, oppure scrivi "
          + "\"fine\" per terminare il programma");
        while (!(stringa = scanner.next()).equals("fine")) {
            stringa = moderaStringa(stringa);
            System.out.println("Hai digitato " + stringa.toUpperCase() + "!");
        }
        System.out.println("Fine programma!");
    }

    private static String moderaStringa(String stringa) {
        switch (stringa) {
            case "accipicchiolina":
            case "perbacco":
            case "stupidino":
            case "giulivo":
            case "giocondo":
            case "perdindirindina":
                stringa = "CENSURATA!";
                break;
            default:
                break;
        }
```

```
            return stringa;
        }
    }
```

È solo una delle soluzioni (non di certo la più elegante).

Soluzione 4.n)

Elenchiamo tutti i casi:

1. `+` se inteso come operatore di addizione no, ma come operatore di concatenazione di stringhe permette di concatenare una stringa a un booleano.
2. `%` no.
3. `++` no.
4. `/=` no.
5. `&` sì, per esempio `(true & false) = true`.
6. `=` sì, è un operatore di assegnazione, applicabile a qualsiasi tipo.
7. `!!` non è un operatore!
8. `>>>` no.

Soluzione 4.o)

La risposta corretta è: `40`.
Infatti `i` inizialmente vale `99`. Poi nella condizione del primo `if`, viene usato un operatore di post-incremento, il quale avendo minore priorità rispetto all'operatore `>=` viene eseguito dopo di esso. Questo implica che `i` vale ancora `99` quando viene testato se è `>= 100`, e solo dopo il test viene incrementato a `100`. Quindi la condizione dell'`if` è `false`, e non viene eseguito il relativo blocco di codice. Quindi viene eseguito il blocco di codice della clausola `else`. Qui viene stampato il risultato di un operatore ternario. Infatti viene decrementato il valore di `i` da `100` a `99`, e quindi l'operatore ternario ritorna il primo valore, ovvero `i++`. Anche in questo caso si tratta di un operatore di post-incremento (a bassa priorità), e quindi viene prima stampato il valore di `i` (`99`) e poi incrementata la variabile `\` (ma tanto il programma termina subito dopo).

Soluzione 4.p)

La risposta corretta è: `32` e al rigo successivo `2`, ovvero l'output è il seguente:

```
32
2
```

Infatti inizialmente i vale `22`, e `j` vale quanto il resto di `22` (e non `23` perché il post-incremento viene applicato dopo l'operatore modulo `%`) diviso `3`, ovvero `1`. Dopodiché ad `i` viene assegnato il valore di ritorno dell'operatore ternario che controlla se `j!=0` (e lo è). Quindi viene ritornato il valore di `j` che è `1`. Nel costrutto `switch` si entra nel `case 1` dove viene stampato `8<<2`, che equivale a `8` moltiplicato per `2` alla seconda, ovvero `32`. Inoltre viene eseguito anche il `case 2`, visto che non c'è un `break` che fa uscire dal costrutto. Quindi viene stampato `8>>2` che equivale a `8` diviso `2` alla seconda, ovvero `2`.

Soluzione 4.q)

Il listato potrebbe essere il seguente:

```
import java.util.Scanner;

public class Esercizio4Q {

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Digita il numero di settimane passate "
          + "dall'ultima vacanza");
        int giorni = scanner.nextInt();
        System.out.println("Hai digitato " + giorni + " giorni!");
        int ore = giorni*24;
        int minuti = ore*60;
        System.out.println("Quindi sono passati " + minuti + " minuti!");
    }
}
```

Soluzione 4.r)

La soluzione potrebbe essere la seguente:

```
public class Esercizio4R {

    private static int matrice[][] = {
        {1, 7, 3, 9, 5, 3},
        {6, 2, 3},
        {7, 5, 1, 4, 0},
        {1, 0, 2, 9, 6, 3, 7, 8, 4}
    };

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

```
        PRIMA_LABEL:
        for (int i = 0; i < matrice.length; i++) {
            int[] riga = matrice[i];
            for (int j = 0; j < riga.length; j++) {
                if (riga[j] == 0) {
                    System.out.println(numeroDaTrovare + " trovato a riga = "
                        + ++i + ", colonna = " + ++j);
                    break PRIMA_LABEL;
                }
            }
        }
        System.out.println("Ricerca terminata");
    }
}
```

Abbiamo dapprima dovuto convertire `args[0]` tramite il metodo statico della classe `Integer` `parseInt()` (cfr. esercizio 3.t) immagazzinandolo in una variabile `numeroDaTrovare`. Poi abbiamo utilizzato un doppio ciclo innestato per navigare all'interno delle celle della matrice, sfruttando gli indici `i` (per le righe) e `j` (per le colonne). Si noti l'utilizzo della label che abbiamo chiamato `PRIMA_LABEL`, che marca il ciclo esterno. Quando viene trovata la prima occorrenza del numero da trovare, l'istruzione:

```
break PRIMA_LABEL;
```

fa terminare anche il ciclo esterno, e il programma continua stampando il messaggio `Ricerca terminata`, e poi termina.

Soluzione 4.s)

Il listato potrebbe essere il seguente:

```
public class Esercizio4S {

    private static int matrice[][] = {
        {1, 7, 3, 9, 5, 3},
        {6, 2, 3},
        {7, 5, 1, 4, 0},
        {1, 0, 2, 9, 6, 3, 7, 8, 4}
    };

    public static void main(String args[]) {
        int numeroDaTrovare = controllaArgomento(args);
        if (numeroDaTrovare == -1) {
            System.out.println("Specificare un numero intero compreso tra"
              + " 0 e 9");
            return;
        }
```

```
        PRIMA_LABEL:
        for (int i = 0; i < matrice.length; i++) {
            int[] riga = matrice[i];
            for (int j = 0; j < riga.length; j++) {
                if (riga[j] == numeroDaTrovare) {
                    System.out.println(numeroDaTrovare + " trovato a riga = "
                      + ++i + ", colonna = " + ++j);
                    break PRIMA_LABEL;
                }
            }
        }
        System.out.println("Ricerca terminata");
    }

    private static int controllaArgomento(String[] args) {
        if (args.length == 1) {
            if (args[0].length() == 1) {
                for (int i = 0; i < 10; i++) {
                    if (args[0].equals("" + i)) {
                        return Integer.parseInt(args[0]);
                    }
                }
            }
        }
        return -1;
    }
}
```

Si noti che abbiamo delegato al metodo `controllaArgomento()` la correttezza dell'input specificato dall'utente. Tale metodo ritorna il valore specificato, oppure, nel caso non sia corretto, il valore `-1`. Come è possibile vedere dal codice del metodo `main()`, nel caso il valore ritornato dal metodo `controllaArgomento()` sia `-1`, viene stampato un messaggio di istruzioni per l'utente, e con l'istruzione `return`, si esce dal metodo. Si noti che il metodo `main()` ritorna `void`, quindi per uscire dal metodo si usa l'istruzione `return` senza specificare cosa ritornare.
Analizziamo ora il metodo `controllaArgomento()`. Con il primo `if` abbiamo dapprima controllato che la lunghezza dell'array `args` sia `0`, ovvero che sia stato specificato un unico argomento, utilizzando la variabile `length` dell'array (cfr. paragrafo 3.6.3). Con il secondo `if` abbiamo controllato che la lunghezza della stringa `args[0]` sia esattamente `1`. Abbiamo sfruttato la chiamata al metodo `length()` della classe `String` (da non confondersi con la variabile `length` dell'array). Il ciclo `for` seguente esegue un ciclo su valori che vanno da `0` a `9`, e controlla che `args[0]` coincida con uno dei valori. Nel momento in cui trova una corrispondenza, il valore corrente viene ritornato dopo averlo convertito a intero mediante la chiamata al metodo statico della classe `Integer` `parseInt()` (cfr. esercizio 3.t). Se invece

nel ciclo `for` non si trovano corrispondenze, per esempio perché è stata specificata una lettera o un simbolo (quindi non un intero compreso tra `0` e `9`), allora il ciclo terminerà e sarà ritornato il valore `-1`.

Soluzione 4.t)

La classe `CalcolatriceSemplificata` richiesta potrebbe essere simile alla seguente:

```
public class CalcolatriceSemplificata {

    public double somma(double d1, double d2) {
        return d1 + d2;
    }

    public double sottrai(double d1, double d2) {
        return d1 - d2;
    }

    public double moltiplica(double d1, double d2) {
        return d1 * d2;
    }

    public double dividi(double d1, double d2) {
        return d1 / d2;
    }

    public double restituisciResto(double d1, double d2) {
        return d1 % d2;
    }

    public double massimo(double d1, double d2) {
        return d1 > d2 ? d1 : d2;
    }

    public double minimo(double d1, double d2) {
        return d1 > d2 ? d2 : d1;
    }

}
```

Mentre la classe per il suo test potrebbe essere:

```
public class Esercizio4T {

    public static void main(String args[]) {
        CalcolatriceSemplificata calcolatriceSemplificata =
          new CalcolatriceSemplificata();
        System.out.println("42.7 + 47.8 = " +
          calcolatriceSemplificata.somma(42.7, 47.8));
```

```
        System.out.println("42.7 - 47.8 = " +
          calcolatriceSemplificata.sottrai(42.7, 47.8));
        System.out.println("42.7 x 47.8 = " +
          calcolatriceSemplificata.moltiplica(42.7, 47.8));
        System.out.println("42.7 : 47.8 = " +
          calcolatriceSemplificata.dividi(42.7, 47.8));
        System.out.println("il resto della divisione tra 42.7 e 47.8 è " +
          calcolatriceSemplificata.restituisciResto(42.7, 47.8));
        System.out.println("Il massimo tra 42.7 e 47.8 è " +
          calcolatriceSemplificata.massimo(42.7, 47.8));
        System.out.println("Il minimo tra 42.7 e 47.8 è " +
          calcolatriceSemplificata.minimo(42.7, 47.8));
    }
}
```

Soluzione 4.u)

Il listato richiesto potrebbe essere il seguente:

```
import java.util.*;

public class Esercizio4U {

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Digita il primo operando e batti enter.");
        double primoOperando = Double.parseDouble(scanner.nextLine());
        System.out.println(
          "Ora scegli l'operazione da eseguire e batti enter:");
        stampaTabellaOperazioni();
        String operazione = scanner.nextLine();
        System.out.println("Ora scegli il secondo operando e batti enter ");
        double secondoOperando = Double.parseDouble(scanner.nextLine());
        double risultato =
          eseguiOperazione(primoOperando, secondoOperando, operazione);
        System.out.println("Risultato = " + risultato);
    }

    private static double eseguiOperazione(double primoOperando,
      double secondoOperando, String operazione) {
        CalcolatriceSemplificata calcolatriceSemplificata =
          new CalcolatriceSemplificata();
        switch (operazione) {
            case "+":
                return calcolatriceSemplificata.somma(
                            primoOperando, secondoOperando);
            case "-":
                return calcolatriceSemplificata.sottrai(
                            primoOperando, secondoOperando);
            case "x":
```

```
                return calcolatriceSemplificata.moltiplica(
                            primoOperando, secondoOperando);
            case "d":
                return calcolatriceSemplificata.dividi(
                            primoOperando, secondoOperando);
            case "r":
                return calcolatriceSemplificata.restituisciResto(
                            primoOperando, secondoOperando);
            case "u":
                return calcolatriceSemplificata.massimo(
                            primoOperando, secondoOperando);
            case "m":
                return calcolatriceSemplificata.minimo(
                            primoOperando, secondoOperando);
            default:
                System.out.println("Operazione specificata " + operazione +
                                   " non valida");
                System.exit(1);
                return Double.NaN;
        }
    }

    private static void stampaTabellaOperazioni() {
        System.out.println("'+' : somma");
        System.out.println("'-' : sottrai");
        System.out.println("'x' : moltiplica");
        System.out.println("'d' : dividi");
        System.out.println("'r' : restituisci resto della divisione");
        System.out.println("'u' : massimo");
        System.out.println("'m' : minimo");
    }
}
```

La traccia dell'esercizio aveva già delineato come implementare l'interazione tra l'utente e il programma. Nel prossimo capitolo vedremo, tra le altre cose, come trovare delle soluzioni progettuali ai nostri programmi, e quanto è importante farlo prima di iniziare a programmare. Per il resto il codice è abbastanza chiaro, e faremo giusto delle osservazioni sui punti più oscuri.

Si noti che per eseguire i calcoli abbiamo usato il tipo di dato numerico più largo: il `double`. Si noti anche l'utilizzo del metodo statico `parseDouble()` della classe `Double`, che converte una stringa in `double` (cfr. documentazione ufficiale), così come il metodo `parseInt()` della classe `Integer` converte una stringa in `int` (cfr. esercizio 3.t).

Non abbiamo gestito (non era richiesto) l'eventuale scorretto utilizzo del programma da parte dell'utente, perché troppo impegnativo per le nozioni che abbiamo studiato sin qui. Con la gestione delle eccezioni che studieremo nel capitolo 9 invece

si troveranno delle soluzioni semplici. Purtroppo esso in alcune operazioni perde di precisione. Potete verificarlo per esempio eseguendo una operazione di resto dalla divisione tra `2.3` e `2`, che dà per risultato un valore non preciso come si può notare da questo esempio di output:

```
Digita il primo operando e batti enter.
2.3
Ora scegli l'operazione da eseguire e batti enter:
'+' : somma
'-' : sottrai
'x' : moltiplica
'd' : dividi
'r' : restituisci resto della divisione
'u' : massimo
'm' : minimo
Ora scegli il secondo operando e batti enter
2
Risultato = 0.2999999999999998
```

Questo è dovuto dal modo in cui viene rappresentato il tipo `double` in memoria, e lo avevamo anche accennato nel capitolo 3.3.2 (bisognerebbe utilizzare il tipo `BigDecimal`, cfr. documentazione ufficiale). Il metodo `System.exit()` fa terminare il programma istantaneamente. C'è da notare inoltre che il programma propone all'utente di scegliere delle lettere specifiche (vedi metodo `eseguiOperazione()`) per eseguire la scelta dell'operazione da eseguire.

Soluzione 4.v)

La classe `StranaCalcolatrice` potrebbe essere la seguente:

```
public class StranaCalcolatrice {

    public double somma(double... doubles) {
        double risultato = 0;
        for (double unDouble : doubles) {
            risultato += unDouble;
        }
        return risultato;
    }

    public double moltiplica(double... doubles) {
        double risultato = doubles[0];
        for (int i = 1; i < doubles.length; i++) {
           risultato *= doubles[i];
        }
        return risultato;
    }
```

```
    public double massimo(double... doubles) {
        double max = doubles[0];
        for (int i = 1; i < doubles.length; i++) {
            double unDouble = doubles[i];
            if (unDouble > max) {
                max = unDouble;
            }
        }
        return max;
    }

    public double minimo(double... doubles) {
        double min = doubles[0];
        for (int i = 1; i < doubles.length; i++) {
            double unDouble = doubles[i];
            if (unDouble < min) {
                min = unDouble;
            }
        }
        return min;
    }
}
```

Abbiamo usato dei varargs come parametri per rendere più agevole la chiamata di questi metodi. Il metodo `somma()` è molto semplice e viene usato un semplice ciclo foreach per eseguire la somma. Negli altri tre casi, abbiamo dovuto prima recuperare il primo elemento, per poi svolgere le operazioni con il resto degli elementi dell'array passato in input.
Mentre la classe di test potrebbe essere la seguente:

```
public class Esercizio4V {

    public static void main(String args[]) {
        StranaCalcolatrice stranaCalcolatrice = new StranaCalcolatrice();
        System.out.println("42.7 + 47.8 = " +
                           stranaCalcolatrice.somma(42.7, 47.8));
        System.out.println("42.7 x 47.8 x 2= " +
          stranaCalcolatrice.moltiplica(42.7, 47.8, 2));
        System.out.println("Il massimo tra 42.7, 47.8, 50, 2, 8, 89 è " +
          stranaCalcolatrice.massimo(42.7, 47.8, 50, 2, 8, 89));
        System.out.println("Il minimo tra 42.7, 47.8, 50, 2, 8, 89 è " +
          stranaCalcolatrice.minimo(42.7, 47.8, 50, 2, 8, 89));
    }
}
```

Soluzione 4.z)

Il listato richiesto potrebbe essere il seguente:

```
import java.util.*;

public class Esercizio4Z {

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        System.out.println(
          "Digita un operando e batti enter, tutte le volte che vuoi." +
          "\nQuando hai finito scegli l'operazione da eseguire e batti enter.");
        stampaTabellaOperazioni();
        String temp;
        String operandiString = "";
        while (isNotOperation(temp = scanner.nextLine())) {
            operandiString += temp + "-";
        }
        if (isNotOperation(temp)) {
            System.out.println("Errore codice operazione");
        }
        String[] operandiArray = operandiString.split("-");
        double[] operandi = new double[operandiArray.length];
        for (int i = 0; i < operandiArray.length; i++) {
            operandi[i] = Double.parseDouble(operandiArray[i]);
        }
        double risultato = eseguiOperazione(operandi, temp);
        System.out.println("Risultato = " + risultato);
    }

    private static boolean isNotOperation(String line) {
        if (line.equals("+") || line.equals("x") ||
            line.equals("u") || line.equals("m")) {
            return false;
        }
        return true;
    }

    private static double eseguiOperazione(double[] operandi,
      String operazione) {
        StranaCalcolatrice stranaCalcolatrice = new StranaCalcolatrice();
        switch (operazione) {
            case "+":
                return stranaCalcolatrice.somma(operandi);
            case "x":
                return stranaCalcolatrice.moltiplica(operandi);
            case "u":
                return stranaCalcolatrice.massimo(operandi);
            case "m":
                return stranaCalcolatrice.minimo(operandi);
```

```
                default:
                    System.out.println("Operazione specificata " + operazione
                      + " non valida");
                    System.exit(1);
                    return Double.NaN;
            }
        }

        private static void stampaTabellaOperazioni() {
            System.out.println("'+' : somma");
            System.out.println("'x' : moltiplica");
            System.out.println("'u' : massimo");
            System.out.println("'m' : minimo");
        }
    }
```

Abbiamo deciso di far specificare all'utente tutti gli operandi, per poi eseguire l'operazione nel momento in cui viene specificato un possibile operatore. Progettare una soluzione al problema non era affatto banale.
Anche in questo caso non abbiamo gestito (non era richiesto) l'eventuale scorretto utilizzo del programma da parte dell'utente, perché troppo impegnativo per le nozioni che abbiamo studiato sin qui. Con la gestione delle eccezioni che studieremo nel capitolo 9 invece si troveranno delle soluzioni semplici.
Il punto critico del codice riguarda la gestione dell'input, che deve essere analizzato e trasformato in tipi di dati che servono per compiere le nostre operazioni (`double`). Purtroppo ci manca un argomento molto importante che non abbiamo ancora studiato come le **collection** (che approfondiremo solo nel capitolo 17, ma introdurremo anche nei capitoli 8 e 11). Senza questo argomento, per immagazzinare i vari operandi specificati dall'utente, ci siamo dovuti arrangiare con una stringa (`operandiString`) che conteneva i vari operandi separati dal simbolo di trattino. Poi con il metodo `split()` (vedi documentazione della classe `String`), abbiamo ottenuto un array di operandi sotto forma di stringa (`operandiArray`). Poi abbiamo istanziato un array di `double operandi` della stessa dimensione di `operandiArray`, e lo abbiamo riempito con gli operandi di tipo `double` dopo averli convertiti tramite il metodo statico `parseDouble()` della classe `Double`.
Una soluzione piuttosto artificiosa, che però ha funzionato.
Ecco un esempio di esecuzione dell'applicazione:

```
Digita un operando e batti enter, tutte le volte che vuoi.
Quando hai finito scegli l'operazione da eseguire e batti enter.
'+' : somma
'x' : moltiplica
'u' : massimo
'm' : minimo
```

```
2
3
4
5
6
7.28
x
Risultato = 5241.6
```

Esercizi del capitolo 5

Sviluppare realmente con Java

Qui troverete tanti esercizi un po' diversi dai soliti esercizi di programmazione. Imparare a programmare ad oggetti è un compito arduo, ma una volta entrati nella mentalità si possono ottenere dei risultati impensabili.

Esercizio 5.a) JShell, Vero o Falso:

1. JShell è un programma che si trova nella directory bin del JDK, come i comandi java e javac, e lo possiamo richiamare direttamente da riga di comando, perché abbiamo impostato correttamente la variabile PATH alla cartella bin a cui appartiene.
2. JShell è un IDE.
3. In una sessione JShell non è possibile dichiarare package.
4. Il simbolo di terminazione di uno statement `;` si può omettere solo se scriviamo un'istruzione Java su di un'unica riga.
5. Dichiarare due volte la stessa variabile è possibile, perché JShell segue regole diverse rispetto al compilatore. L'ultima variabile dichiarata sovrascriverà la precedente.
6. Se dichiariamo una variabile di tipo `String` senza inizializzarla, essa sarà inizializzata automaticamente a `null`.
7. Non è possibile dichiarare un'interfaccia in una sessione JShell.

8. Il modificatore `abstract` è sempre ignorato all'interno di una sessione JShell.
9. Non è possibile dichiarare un metodo main() in una sessione JShell.
10. È possibile dichiarare annotazioni ed enumerazioni in una sessione JShell.

Esercizio 5.b) JShell comandi, Vero o Falso:

1. Per terminare una sessione JShell, bisogna digitare il comando `goodbye`.
2. Tutti i comandi di JShell devono avere come prefisso il simbolo `\`.
3. I comandi `help` e `?` sono equivalenti.
4. Il comando `history` mostra tutti gli snippet e tutti i comandi eseguiti dall'utente nella sessione corrente. Accanto agli snippet è presente uno snippet id che permette di richiamare lo snippet corrispondente.
5. Il comando `list` mostra tutti i comandi digitati dall'utente in questa sessione.
6. Il comando `/types -all` elencherà tutte le variabili dichiarate nella sessione corrente.
7. Il comando `reload` provocherà che tutte le istruzioni eseguite nella sessione corrente siano eseguite nuovamente.
8. Il comando `drop` può cancellare un certo snippet specificando il suo snippet id.
9. Se specifichiamo il comando `/set start JAVASE`, importeremo in questa ed in tutte le altre sessioni future di JShell, tutte i package di Java Standard Edition.
10. Con il comando `/!` richiamiamo l'ultimo snippet editato, sia esso valido o non valido.

Esercizio 5.c)

Consideriamo le seguenti righe editate all'interno di una sessione JShell:

```
jshell> public int a;
a ==> 0

jshell> private String a
```

```
a ==> null

jshell> /reset
|  Resetting state.


jshell> /list
```

Quale sarà l'output del comando `list` finale?

Esercizio 5.d)

Considerando tutte le istruzioni del precedente esercizio, quale sarà l'output del seguente comando?

```
jshell> /history
```

Esercizio 5.e)

Se volessimo aprire il file HelloWorld.java (realizzato nel primo capitolo) all'interno di JShell che si trova nella cartella corrente, che comando dobbiamo eseguire?

1. `/save HelloWorld.java`
2. `/retain HelloWorld.java`
3. `/reload HelloWorld.java`
4. `/open HelloWorld.java`
5. `/start HelloWorld.java`
6. `/env HelloWorld.java`
7. `/!`

Esercizio 5.f)

Una volta aperto il file HelloWorld.java all'interno di una sessione JShell, come possiamo far stampare la stringa `Hello World!`?

Esercizio 5.g)

Che comando dobbiamo eseguire se vogliamo copiare il file HelloWorld.java aperto nell'esercizio 5.e in una cartella `C:/cartella`?

1. `/save HelloWorld.java`

2. `/retain HelloWorld.java`
3. `/save HelloWorld2.java`
4. `/save C:/cartella/HelloWorld.java`
5. `/save -start start C:/cartella/HelloWorld.java`
6. `/env C:/cartella/HelloWorld.java`
7. `/! C:/cartella/HelloWorld.java`

Esercizio 5.h) JShell strumenti ausiliari, Vero o Falso:

1. In una sessione JShell è possibile dichiarare una variabile senza specificare un reference.
2. In una sessione JShell è possibile dichiarare un reference senza specificare il tipo del reference.
3. In una sessione JShell è possibile scrivere un valore, poi con il tasto TAB far dedurre automaticamente il tipo della variabile a JShell, per poi scrivere il nome del reference.
4. Mentre si dichiara un reference ad un tipo non importato, è possibile farci suggerire da JShell una lista di possibili `import` da usare per il tipo, premendo contemporaneamente i tasti SHIFT e TAB, rilasciarli e dopo premere il tasto v.
5. Il comando `/edit` aprirà il programma Notepad++.
6. In una sessione JShell la pressione contemporanea dei tasti CTRL - BARRA SPAZIATRICE provoca l'auto-completamento del codice che si è iniziato a scrivere, o nel caso di più opzioni disponibili, la scelta tra esse.
7. In una sessione JShell la pressione contemporanea dei tasti CTRL - E provoca lo spostamento del cursore al termine della riga.
8. In una sessione JShell la pressione contemporanea dei tasti ALT - D provoca la cancellazione della parola alla destra del cursore.
9. Il comando `/set feedback silent` provoca che JShell eviti di stampare i messaggi dell'analisi del codice scritto.
10. Scrivendo `System` e premendo due volte il tasto TAB, JShell ci mostrerà la documentazione della classe `System`.

Esercizio 5.i)

L'utilizzo di un IDE implica (scegliere tutte le risposte che si ritiene siano corrette):

1. Iniziale rallentamento dell'apprendimento dello sviluppo, perché richiede lo studio dello stesso IDE.
2. La possibilità di utilizzare un debugger. Fondamentale strumento per sviluppare.
3. La possibilità di integrarsi con altri strumenti di sviluppo con un database o un application server.
4. La possibilità di utilizzare un editor evoluto che ci permette di automatizzare la creazione di costrutti di programmazione ed utilizzare tecniche di refactoring del codice.
5. L'impossibilità di utilizzare i package, che devono comunque essere gestiti da riga di comando.

Esercizio 5.l)

L'architettura si occupa (scegliere tutte le risposte che si ritiene siano corrette):

1. Di definire diagrammi di attività UML che modellano le funzionalità del sistema.
2. Di migliorare le prestazioni del software.
3. Di ottimizzare l'utilizzo delle risorse da parte del software.
4. Di ottimizzare il partizionamento del software in modo tale da semplificare la procedura di installazione.

Esercizio 5.m)

Un deployment diagram (scegliere tutte le affermazioni che si ritiene siano corrette):

1. È un diagramma statico.
2. Mostra i flussi di esecuzione dell'applicazione.
3. Mostra come i nodi hardware ospitino i componenti software.

4. Evidenziano le dipendenze tra componenti software.
5. Il principale elemento del diagramma è un *nodo*, che è rappresentato da un rettangolo con due piccoli rettangoli che fuoriescono dall'angolo superiore sinistro.

Esercizio 5.n)

Abbiamo asserito che la conoscenza di base di argomenti quali XML e database è essenziale per lavorare in un'azienda informatica. Questo perché la quasi totalità delle applicazioni utilizza in qualche modo queste due tecnologie. Se si lavora ad un'applicazione web, quali sono le conoscenze di base che bisogna avere?

Esercizio 5.o)

Definire brevemente i concetti di client, server, applicazione standalone, mobile, web, client web, server web ed applicazione enterprise.

Esercizio 5.p)

Definire i compiti dei seguenti ruoli aziendali:

1. Capo progetto
2. Business analyst
3. IT manager
4. Release manager
5. DBA
6. Grafico

Esercizio 5.q)

Definire cos'è una metodologia object oriented.

Esercizio 5.r)

Problem statement: creare un programma di autenticazione che richiede all'utente username e password, e gli garantisce l'accesso (basterà stampare un messaggio di benvenuto con il nome dell'utente) se le credenziali inserite sono corrette. Il sistema deve supportare l'autenticazione con almeno tre coppie username e password, con cui ci si può autenticare.
Eseguire l'analisi dei casi d'uso ed individuare i vari casi d'uso, seguendo i consigli dei

paragrafi 5.4.1 e 5.4.1.1. Esistono vari strumenti software che supportano i diagrammi UML (**https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools**). Ognuno di essi ha la sua filosofia e la sua complessità. Dovreste sceglierne uno prima o poi (magari dopo aver fatto qualche confronto), anche se per ora va benissimo usare un foglio di carta, una matita e una gomma da cancellare. La scelta di uno strumento, capire come funziona, etc. richiede del tempo.

Esercizio 5.s)

Individuare gli scenari dei casi d'uso del precedente esercizio seguendo i consigli dei paragrafi 5.4.1.2.

Esercizio 5.t)

Continuiamo l'esercizio precedente seguendo il processo descritto nel paragrafo 5.4.2. Essendo una semplice applicazione desktop, sembra superfluo creare un diagramma di deployment. Ma se pensiamo a come poter un giorno riutilizzare una parte di quest'applicazione (d'altronde stiamo parlando di un'applicazione che permette di autenticarsi in un sistema, e che potremmo integrare anche nel caso di studio introdotto nel capitolo 5 per autenticarsi nell'applicazione *Logos*), allora potrebbe essere molto utile realizzare una visione dall'alto che ne specifica le dipendenze tra i vari componenti del software. Quindi proviamo a creare un component diagram (o volendo un deployment diagram) creando componenti che abbiano nomi indicativi, e che poi provvederemo a realizzare con le classi. Deve essere giusto un banale diagramma (high level deployment diagram) che esalti la possibilità che una certa parte del software sia riutilizzabile.

Esercizio 5.u)

Continuiamo l'esercizio precedente seguendo il processo descritto nel paragrafo 5.4.3. Passiamo quindi ad individuare le classi candidate, e di conseguenza le key abstraction.

Esercizio 5.v)

Continuiamo l'esercizio precedente verificando la fattibilità degli scenari individuati, creando sequence diagram basati sulle interazioni delle classi individuate, come descritto nel paragrafo 5.4.4.

Esercizio 5.z)

In base ai passi fatti negli esercizi 5.r, 5.s, 5.t, 5.u, e 5.v, implementare una soluzione funzionante.

Soluzioni degli esercizi del capitolo 5

Soluzione 5.a) JShell, Vero o Falso:

1. **Vero**.
2. **Falso**.
3. **Vero**.
4. **Vero**.
5. **Vero**.
6. **Vero**, le variabili non inizializzate sono inizializzate ai propri valori nulli. Una stringa, essendo un oggetto, ha come valore nullo proprio `null`.
7. **Falso**.
8. **Falso**.
9. **Falso**, è possibile dichiarare un metodo `main()`, ma non avrà lo stesso ruolo di "metodo iniziale" come in un ordinario programma Java.
10. **Vero**.

Soluzione 5.b)

1. **Falso**, bisogna digitare il comando `exit`.

2. **Falso**, tutti i comandi JShell devono avere come prefisso `/`.
3. **Vero**.
4. **Falso**, è vero che il comando `history` mostra tutti gli snippet e tutti i comandi eseguiti dall'utente nella sessione corrente, ma non è vero che accanto agli snippet è presente uno snippet id.
5. **Falso**, il comando `list` mostra tutti gli snippet digitati dall'utente in questa sessione, con accanto i rispettivi snippet id.
6. **False**, il comando `/types -all` elencherà tutte i tipi (classi, interfacce, enumerazioni, annotazioni) dichiarate nella sessione corrente. Piuttosto il comando `/variables -all` elencherà tutte le variabili dichiarate nella sessione corrente.
7. **Vero**.
8. **Vero**.
9. **Falso**, è vero che importeremo in questa sessione, tutte i package di Java Standard Edition, ma non nelle future sessioni (avremmo dovuto esplicitare anche l'opzione `-retain`).
10. **Vero**.

Soluzione 5.c)

L'output del comando `list` sarà vuoto. Infatti il comando `reset` avrà reimpostato tutti gli snippet immessi.

Soluzione 5.d)

L'output del comando `history` sarà il seguente:

```
public int a;
private String a
/reset
/list
/history
```

Soluzione 5.e)

Il comando corretto è:

4. `/open HelloWorld.java`

Soluzione 5.f)

Possiamo solo invocare il metodo `main()` nel seguente modo:

```
jshell> HelloWorld hw = new HelloWorld();
hw ==> HelloWorld@52a86356

jshell> hw.main(null);
Hello World!
```

Si noti che siccome l'array `args` non viene utilizzato all'interno del metodo `main()`, allora abbiamo potuto passargli `null`.
Anche se non l'abbiamo ancora studiato seriamente, il modificatore `static` ci permette di evitare di istanziare l'oggetto `hw`, e di eseguire direttamente il comando usando il nome della classe:

```
jshell> HelloWorld.main(null)
Hello World!
```

Soluzione 5.g)

Il comando corretto è:

4. `/save C:/cartella/HelloWorld.java`

Soluzione 5.h)

1. **Vero**, in questo caso si parla di variabile implicita, e JShell ne dedurrà automaticamente il tipo.
2. **Vero**, in questo caso si parla di forwarding reference, e JShell creerà il reference, ma non lo renderà disponibile sino a quando non dichiareremo anche il suo tipo. A quel punto il reference sarà sostituito e inizializzato a `null`.
3. **Falso**, bisogna invece premere contemporaneamente i tasti SHIFT e TAB, rilasciarli e dopo premere il tasto v (che sta per "variable"). JShell dedurrà il tipo della variabile, la dichiarerà e posizionerà il cursore subito dopo per permetterci di definire il reference.
4. **Falso**, bisogna invece premere contemporaneamente i tasti SHIFT e TAB, rilasciarli e dopo premere il tasto i (che sta per "input").

5. **Falso**, verrà aperto il programma JShell Edit Pad, a meno che prima non si sia impostato come editor di default Notepad++ mediante il comando:

```
/set editor C:\Program Files (x86)\Notepad++\notepad++.exe
```

6. **Falso**, la pressione del tasto TAB provoca l'auto-completamento del codice che si è iniziato a scrivere, o nel caso di più opzioni disponibili, la scelta tra esse.
7. **Vero**.
8. **Vero**.
9. **Vero**.
10. **Vero**.

Soluzione 5.i)

Solo la quinta risposta è falsa. Un IDE invece esonera il programmatore dalla complessa gestione dei package.

Soluzione 5.l)

Solo la prima risposta è falsa. Un architetto infatti si dedica essenzialmente ai requisiti non funzionali dell'applicazione.

Soluzione 5.m)

Sono errate le affermazioni:

- numero 2: perché non sono evidenziati flussi di esecuzione.
- numero 5: è vero che il nodo è il principale elemento del diagramma, ma viene rappresentato come un cubo tridimensionale. La descrizione riportata invece si riferisce all'elemento *componente*.

Soluzione 5.n)

Se si lavora nel campo web avere delle conoscenze basiche sul protocollo HTTP, i linguaggi HTML, Javascript e CSS, librerie come Bootstrap, e framework come Spring, o Angular JS e così via.

Soluzione 5.o)

Per definizione un *client* è un programma che richiede servizi ad un altro programma che si chiama server. Per definizione un *server*, è un programma sempre in esecuzione, che mette a disposizione dei servizi. Un client e un server comunicano solitamente tramite rete, con un protocollo ben definito.
Le *applicazioni standalone*, (dette anche *applicazioni desktop*), sono eseguite su computer desktop e portatili, e solitamente hanno un'interfaccia grafica.
Le *applicazioni web* sono applicazioni che hanno un'architettura suddivisa in una parte client e una parte server.
Un *client web* richiede pagine web, e coincide con i programmi che chiamiamo comunemente *browser* (Mozilla Firefox, Google Chrome e così via).
Un *server web* è invece un'applicazione che mette a disposizione servizi disponibili in rete.
Le *applicazioni enterprise* (che potremmo tradurre come *applicazioni aziendali*) sono un'evoluzione delle applicazioni web, e solitamente mettono a disposizione servizi più complessi come scaricamento di risorse, *web services* (ovvero applicazioni di comunicazione tra sistemi eterogenei che usano il protocollo HTTP), servizi di reportistica, etc., e che quindi come *client enterprise* possono avere anche programmi appositamente creati per interagire con lo strato *server enterprise*. Quest'ultimo è costituito a sua volta da vari strati che utilizzano tecnologie diverse per adempiere a vari scopi.
Un'*applicazione mobile* (spesso chiamata semplicemente *app*) è un'applicazione che viene eseguita su client mobili, come smartphone e tablet, e può anche avere una controparte server.

Soluzione 5.p)

Leggi paragrafo 5.3.4.

Soluzione 5.q)

Una *metodologia object oriented*, nella sua definizione più generale, potrebbe intendersi come una coppia costituita da un processo e da un linguaggio di modellazione.
A sua volta un *processo* potrebbe essere definito come la serie di indicazioni riguardanti i passi da intraprendere per portare a termine con successo un progetto.
Un *linguaggio di modellazione* è invece lo strumento che le metodologie utilizzano per descrivere (possibilmente in maniera grafica) tutte le caratteristiche statiche e dinamiche di un progetto. Il linguaggio di modellazione considerato standard de facto è l'UML.

Soluzione 5.r)

Esiste un unico caso d'uso che abbiamo chiamato "Login" (vedi figura 5.r.1). Infatti le interazioni che l'utente avrà con il sistema, si limitano alle attività relative all'autenticazione, come l'inserimento della username e della password. Anche pensando a diverse tipologie di flussi che possono realizzare l'autenticazione, il compito finale è unicamente quello di autenticarsi.

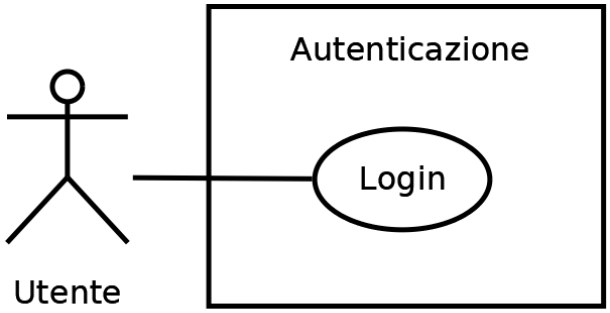

Figura 5.r.1 - Use case diagram.

Nel nostro caso abbiamo trovato un unico caso d'uso, ma questo non significa che sarà sempre così, e che quindi bisogna evitare l'analisi degli use case! Essa è indispensabile anche per i programmi più banali.

Soluzione 5.s)

L'analisi degli scenari è molto soggettiva. Nel momento in cui la scriviamo stiamo decidendo "cosa" deve fare l'applicazione, qualcosa tutt'altro che scontato.
Partiamo dal prerequisito che non abbiamo ancora studiato le interfacce grafiche (a cui sono dedicate due corpose appendici) e quindi abbiamo pensato di realizzare un programma che funzioni solo da riga di comando.
Un altro prerequisito è che il sistema ha precaricato staticamente alcune coppie di username e password validi.

Scenario principale

1. Il sistema chiede all'utente di inserire lo username.
2. L'utente inserisce lo username.

3. Il sistema verifica che lo username sia valido e chiede di inserire la password.
4. L'utente inserisce la password.
5. Il sistema controlla che la password sia valida e risponde con un messaggio di conferma di avvenuta autenticazione, utilizzando il vero nome dell'utente che si è autenticato.

Come già detto questa è solo una delle possibili soluzioni. Potremmo anche pensare di specificare insieme username e password, convalidare l'autenticazione con un codice captcha, avvertire l'utente se il tasto CAPS LOCK (BLOCCO MAIUSCOLO) è inserito, mascherare i caratteri della password con asterischi, richiedere all'utente se vuole memorizzare lo username per il prossimo accesso e così via. Abbiamo scelto un'interazione semplice.

Scenario 2

1. Il sistema chiede all'utente di inserire lo username.
2. L'utente inserisce lo username.
3. Il sistema non riconosce lo username inserito, stampa un messaggio e torna al punto 1.

Scenario 3

1. Il sistema chiede all'utente di inserire lo username.
2. L'utente inserisce lo username.
3. Il sistema verifica che lo username sia valido, e chiede di inserire la password.
4. L'utente inserisce la password.
5. Il sistema non riconosce la password inserita, stampa un messaggio e torna al punto 1.
6. Definendo questi tre banali scenari abbiamo ben chiaro cosa fare.

Soluzione 5.t)

Con il diagramma della figura 5.t.1, evidenziamo come creeremo un componente software che contiene le classi che realizzano l'autenticazione, separato dalla classe del `main()`. L'unico strumento che attualmente conosciamo per separare le classi sono i package, quindi potremo usare package diversi.

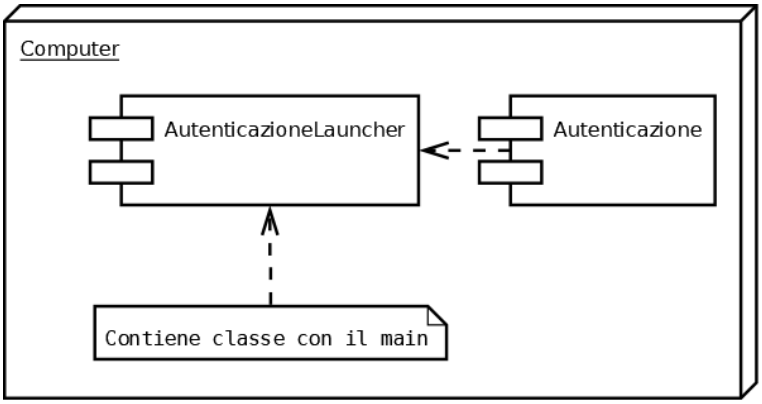

Figura 5.t.1 - Architettura dell'applicazione espressa con un high level deployment diagram.

Consultando l'appendice E potete anche imparare a creare il file JAR contenente le classi del componente dell'autenticazione. In questo modo sarà più semplice poterlo riutilizzare in un secondo momento trattandosi di un unico file. La traccia dell'esercizio E.a relativo a quell'appendice richiede proprio che sia creato il file JAR con le classi di questo esercizio.

Soluzione 5.u)

Seguendo il processo descritto nel paragrafo 5.4.3, per poter trovare la lista delle *key abstraction*, dobbiamo dapprima stilare la lista delle *candidate classes*. Segue la nostra lista con relativi commenti.

1. **Autenticazione**: potrebbe essere una classe a cui diamo la responsabilità di gestire la funzionalità principale dell'applicazione.

2. Login: sembra più il nome del metodo principale dell'applicazione. Potrebbe essere un metodo all'interno di `Autenticazione` o di un oggetto dichiarato all'interno di `Autenticazione`.
3. **Utente**: potrebbe essere l'entità che contiene le informazioni per l'autenticazione.
4. Sistema: troppo generico, non sembra essere il nome giusto per una classe.
5. Username: potrebbe essere una proprietà della classe `Utente`.
6. Password: potrebbe essere una proprietà della classe `Utente`.
7. Nome: potrebbe essere una proprietà della classe `Utente`.
8. Verifica: potrebbe essere un metodo di `Autenticazione`, o di un oggetto dichiarato all'interno di `Autenticazione`.
9. Inserimento: potrebbe definire un metodo, ma non sembra un'astrazione chiave.
10. Messaggio: per ora sembra solo una stringa, non di certo un'astrazione chiave. Questo non esclude che possa diventare una classe in seguito.
11. Sostituzione: potrebbe definire un metodo, ma non sembra un'astrazione chiave.

Quindi la lista delle astrazioni chiave, per ora si limita solo alle uniche due classi della lista in grassetto: `Autenticazione` e `Utente`. A queste aggiungiamo una classe `AutenticazioneLauncher` che contiene solo il metodo `main()` e che ha la sola responsabilità di avviare l'applicazione istanziando gli oggetti corretti e richiamandone i metodi opportunamente.

Ricordiamo al lettore che la scelta di queste classi, come i precedenti passi, dipendono dall'esperienza, dalle intenzioni, dal tempo a disposizione, dalla predisposizione, dalla concretezza e dalla forma mentis della persona che svolge l'analisi. Esistono migliaia di soluzioni che possono portare al successo lo sviluppo, ognuna delle quali ha dei pro e dei contro. Il consiglio è quello di orientarsi (soprattutto nei primi tempi) sulla soluzione più semplice possibile.

Soluzione 5.v)

Seguendo il processo descritto nel paragrafo 5.4.4, adesso dobbiamo dare la definizione superficiale delle key abstraction, con diagrammi di interazione (cfr. appendice L), ovvero collaboration o sequence diagram.

Questi due diagrammi sono equivalenti, tanto che molti tool UML, permettono di trasformare un collaboration diagram in un sequence diagram e viceversa con la pressione di un tasto. In particolare un sequence diagram mostra le interazioni tra gli oggetti in un determinato lasso di tempo, enfatizzando la sequenza dei messaggi che le entità si scambiano. Un collaboration diagram invece, come il sequence diagram mostra le interazioni tra gli oggetti in un determinato lasso di tempo, ma enfatizza l'organizzazione strutturale delle entità che interagiscono.

Siccome in questo caso ci sembra più interessante enfatizzare la sequenza di messaggi scambiati tra gli oggetti, useremo sequence diagram per descrivere gli scenari descritti nella soluzione dell'esercizio 5.s, usando gli oggetti descritti nell'esercizio 5.u. Dato che questi ultimi oggetti sono solo astrazioni chiave, in questo momento possiamo anche decidere se dobbiamo creare nuove classi, aggiungere, modificare o spostare metodi, rinominare le classi esistenti e così via. Il diagramma, con la sua "visione dell'alto", favorisce l'individuazione di eventuali situazioni errate, migliorabili o fallaci.

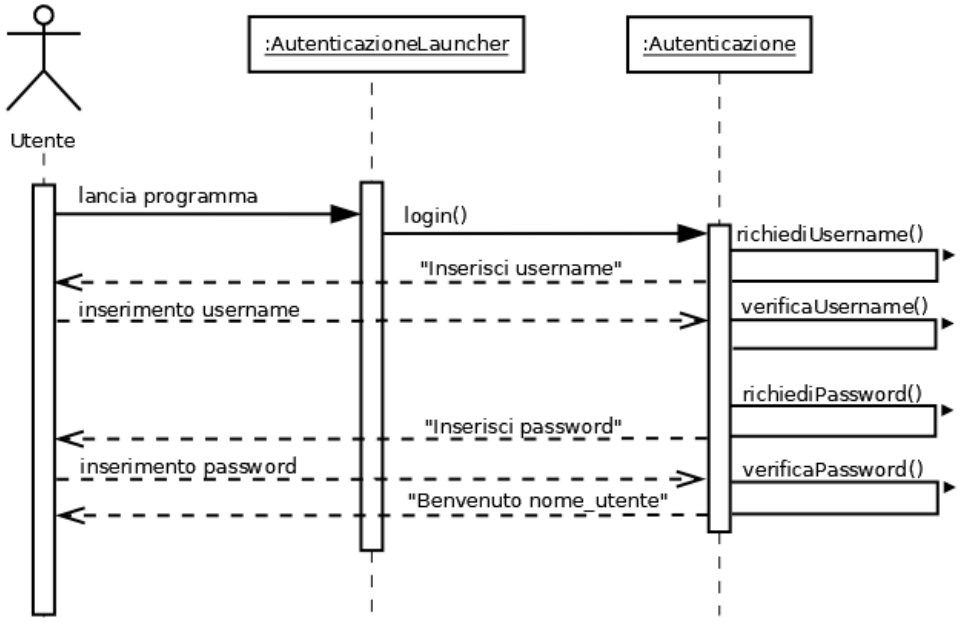

Figura 5.v.1 - Sequence diagram che rappresenta lo scenario principale.

In figura 5.v.1 ci siamo limitati a trovare un flusso coerente con quello descritto nello scenario principale, utilizzando solo le key abstraction individuate nell'esercizio precedente. La situazione vista così, sembra funzionare.
L'utente dell'applicazione esegue l'applicazione tramite la classe `AutenticazioneLauncher`. Questa chiama un metodo che si chiama `login()` sulla classe `Autenticazione`. Da qui in poi questo metodo eseguirà delle operazioni. Prima richiama il metodo `richiediUsername()` che richiede lo username all'utente stampando un messaggio, e aspetta l'input dell'utente.

Questo metodo è interno, cioè definito nella stessa classe. Lo si capisce dal fatto che la freccia che lo definisce parte e termina nella stessa classe. In UML la chiamata di un metodo è indicata dalla partenza di una "freccia" (che come elemento UML viene chiamato messaggio) da un certo oggetto. La "freccia" punta all'oggetto dove risiede il metodo chiamato. Quindi una freccia che ritorna sullo stesso oggetto sta ad indicare la chiamata ad un metodo interno.

Poi una volta che l'utente inserisce lo username, la classe `Autenticazione` chiamerà un metodo interno chiamato `verificaUsername()`. Tale metodo recupererà l'oggetto utente dalla collezione di utenti che è stata definita per rappresentare la lista di utenti (che implementeremo tramite array).

Questa parte non è stata rappresentata nel diagramma, per non complicarlo troppo. Avremmo potuto aggiungere un elemento nota per specificare le nostre intenzioni, sarebbe stato più corretto, ma abbiamo evitato visto che dovevamo spiegare con queste righe il diagramma.

Quindi anche se l'oggetto `Utente` non compare sul diagramma è in qualche modo coinvolto. Questo perché nella nostra mente, un utente dovrebbe definire le variabili `username`, `password` e `nome`, e quindi per verificare se esiste un certo username, ma anche per vedere se una password è associata ad un certo username, bisogna utilizzare un oggetto `Utente`.
Il resto del flusso è molto semplice da interpretare. Viene chiamato il metodo interno `richiediPassword()`, che richiede la password all'utente stampando

un messaggio, e aspetta l'input dell'utente. Poi una volta che l'utente inserisce la password, la classe `Autenticazione` chiamerà un metodo interno chiamato `verificaPassword()` (ed anche in questo caso la verifica verrà fatta utilizzando un oggetto `Utente`). Infine viene stampato il messaggio Benvenuto specificando il nome dell'utente che si è autenticato (che si prende dall'oggetto utente usato per convalidare l'autenticazione).

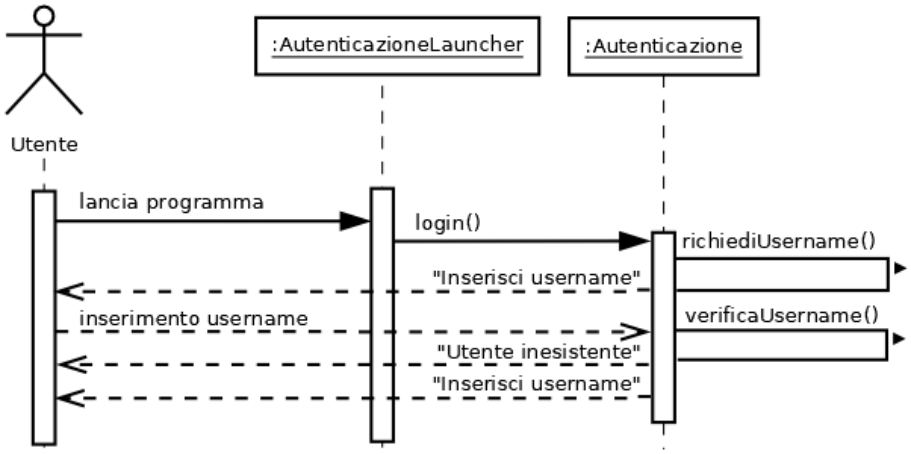

Figura 5.v.2 - Sequence diagram che rappresenta il secondo scenario.

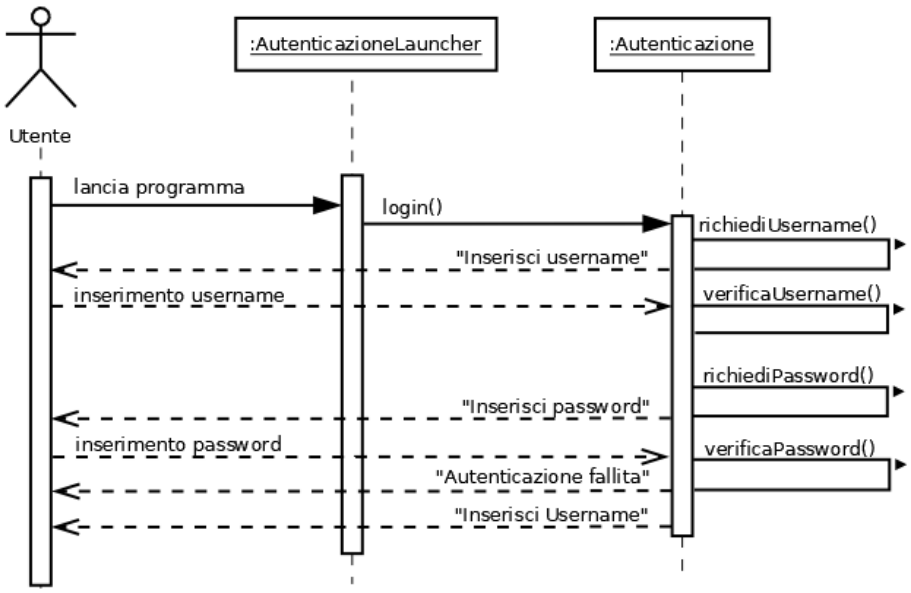

Figura 5.v.3 - Sequence diagram che rappresenta il terzo scenario.

Possiamo notare che per come abbiamo progettato il diagramma in figura 5.v.2, e nello scenario relativo, lo username sarà richiesto sin quando non verrà immesso uno username valido. Stesso discorso per la password, come è possibile osservare in figura 5.v.3 che mostra un sequence diagram relativo al terzo scenario individuato.

Soluzione 5.z)

Il listato che è scaturito dalla nostra analisi non è proprio quello che ci aspettavamo:

```
package com.claudiodesio.autenticazione;
import java.util.*;
public class Autenticazione {
    private static final Utente[] utenti = {
        new Utente("Daniele", "dansap", "musica"),
        new Utente("Giovanni", "giobat", "scienze"),
        new Utente("Ligeia", "ligder", "arte")
    };
    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("Inserisci username.");
            String username = scanner.nextLine();
            Utente utente = verificaUsername(username);
            if (utente == null) {
                System.out.println("Utente non trovato!");
                continue;
            }
            System.out.println("Inserisci password");
            String password = scanner.nextLine();
            if (password != null && password.equals(utente.password)) {
                System.out.println("Benvenuto " + utente.nome);
                break;
            } else {
                System.out.println("Autenticazione fallita");
            }
        }
    }
    private static Utente verificaUsername(String username) {
        if (username != null) {
            for (Utente utente : utenti) {
                if (username.equals(utente.username)) {
                    return utente;
                }
            }
        }
        return null;
    }
}
```

Nel momento in cui abbiamo scritto il codice abbiamo trovato alcune difficoltà. La prima è stata implementare l'algoritmo descritto negli scenari: abbiamo dovuto introdurre un ciclo infinito e i comandi `break` e `continue`, non proprio la soluzione che ci aspettavamo.
Abbiamo preso anche un'altra decisione in contrasto con la nostra analisi: eliminare la classe `AutenticazioneLauncher` che avrebbe dovuto contenere solo il metodo `main()`. Questo perché sembrava una decisione forzata e priva di una qualche utilità.
Un altro dubbio ci è sorto nel momento in cui abbiamo dovuto verificare la correttezza di username e password, in quanto non avevamo deciso in fase di progettazione se la verifica doveva tener conto delle lettere maiuscole o minuscole.
Non siamo riusciti neanche a creare un metodo `verificaPassword()` complementare al `verificaUsername()`, visto che se avessimo creato un metodo a parte non avremmo potuto utilizzare la clausola `break` per uscire dal ciclo infinito. Insomma, rispetto all'analisi che abbiamo fatto, ci sono stati dei problemi che abbiamo risolto direttamente con il codice. Ma da dove sono scaturiti questi problemi? Perché il nostro processo non ha funzionato come speravamo?
La risposta è che essenzialmente ci mancano concetti fondamentali che dobbiamo ancora studiare. Perciò questo esercizio sarà ripreso tra gli esercizi del prossimo capitolo per renderlo più chiaro ed efficiente. In particolare ci sono mancati due passaggi fondamentali: assegnare le responsabilità alle classi che creiamo, e creare un diagramma delle classi che ci aiuti a meglio distribuire le responsabilità tra le classi. Ogni oggetto deve avere un'unica responsabilità, o più responsabilità relazionate strettamente tra loro. Le responsabilità saranno implementate o come metodi o come variabili, e definiscono ruoli che si possono assegnare agli oggetti.
In qualsiasi caso il programma che abbiamo scritto funziona correttamente. Anche se la nostra analisi non è stata perfetta, ci ha dato delle indicazioni importanti. Per esempio l'analisi degli scenari è stata fondamentale per capire cosa dovevamo fare. E i diagrammi di interazione ci hanno anche indirizzato verso la soluzione di implementazione (poi parzialmente disattesa).

Dopo l'esperienza fatta con questi esercizi, nel caso non siate riuscite a risolvere gli esercizi 4.m, 4.q, 4.r, 4.s, 4.t, 4.u, 4.v e 4.z del capitolo precedente, potreste provare a riprogettarli da zero e cercare di trovare una soluzione da soli, magari diversa da quella proposta.

Esercizi del capitolo 6

Incapsulamento e visibilità

È ora di entrare pian piano nella giusta mentalità object oriented.

Dopo ogni esercizio conviene consultare anche la soluzione, perché spesso un esercizio è propedeutico al successivo.

Esercizio 6.a) Object Orientation in generale (teoria), Vero o Falso:

1. L'Object Orientation esiste solo da pochi anni.
2. Java è un linguaggio object oriented non puro, SmallTalk è un linguaggio object oriented puro.
3. Tutti i linguaggi orientati agli oggetti supportano allo stesso modo i paradigmi object oriented. Si può dire che un linguaggio è object oriented se supporta incapsulamento, ereditarietà e polimorfismo; infatti altri paradigmi come l'astrazione e il riuso appartengono anche alla filosofia funzionale.
4. Applicare l'astrazione significa concentrarsi solo sulle caratteristiche importanti dell'entità da astrarre.
5. La realtà che ci circonda è fonte d'ispirazione per la filosofia object oriented.
6. L'incapsulamento ci aiuta ad interagire con gli oggetti, l'astrazione ci aiuta ad interagire con le classi.

7. Il riuso è favorito dall'implementazione degli altri paradigmi object oriented.

8. L'ereditarietà permette al programmatore di gestire in maniera collettiva più classi.

9. L'incapsulamento divide gli oggetti in due parti separate: l'interfaccia pubblica e l'implementazione interna.

10. Per l'utilizzo dell'oggetto basta conoscere l'implementazione interna, non è necessario conoscere l'interfaccia pubblica.

Esercizio 6.b) Incapsulare e completare le seguenti classi:

```
public class Pilota {
    public String nome;

    public Pilota(String nome) {
        // impostare il nome
    }
}

public class Auto {
    public String scuderia;
    public Pilota pilota;

    public Auto(String scuderia, Pilota pilota) {
        // impostare scuderia e pilota
    }
    public String dammiDettagli() {
        // restituire una stringa descrittiva dell'oggetto
    }
}
```

Tenere presente che le classi `Auto` e `Pilota` devono poi essere utilizzate dalle seguenti classi:

```
public class TestGara {
    public static void main(String args[]) {
        Gara imola = new Gara("GP di Imola");
        imola.corriGara();
        String risultato = imola.getRisultato();
        System.out.println(risultato);
    }
}

public class Gara {
    private String nome;
```

```
    private String risultato;
    private Auto griglia [];

    public Gara(String nome) {
        setNome(nome);
        setRisultato("Corsa non terminata");
        creaGrigliaDiPartenza();
    }

    public void creaGrigliaDiPartenza() {
        Pilota uno = new Pilota("Pippo");
        Pilota due = new Pilota("Pluto");
        Pilota tre = new Pilota("Topolino");
        Pilota quattro = new Pilota("Paperino");
        Auto autoNumeroUno = new Auto("Ferrari", uno);
        Auto autoNumeroDue = new Auto("Renault", due);
        Auto autoNumeroTre = new Auto("BMW", tre);
        Auto autoNumeroQuattro = new Auto("Mercedes", quattro);
        griglia = new Auto[4];
        griglia[0] = autoNumeroUno;
        griglia[1] = autoNumeroDue;
        griglia[2] = autoNumeroTre;
        griglia[3] = autoNumeroQuattro;
    }

    public void corriGara() {
        int numeroVincente = (int)(Math.random() * 4);
        Auto vincitore = griglia[numeroVincente];
        String risultato = vincitore.dammiDettagli();
        setRisultato(risultato);
    }

    public void setRisultato(String vincitore) {
        this.risultato = "Il vincitore di " + this.getNome() + ": "
          + vincitore;
    }

    public String getRisultato() {
        return risultato;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }
}
```

Analisi dell'esercizio

La classe `TestGara` contiene il metodo `main()` e quindi determina il flusso di esecuzione dell'applicazione. È molto leggibile: si istanzia un oggetto gara e lo si chiama "GP di Imola", si fa correre la corsa, si richiede il risultato e lo si stampa a video.

La classe `Gara` invece contiene pochi e semplici metodi e tre variabili d'istanza: `nome` (il nome della gara), `risultato` (una stringa che contiene il nome del vincitore della gara se è stata corsa) e `griglia` (un array di oggetti `Auto` che partecipano alla gara). Il costruttore prende in input una stringa con il nome della gara che viene opportunamente impostato. Inoltre il valore della stringa risultato è impostata a "Corsa non terminata". Infine è chiamato il metodo `creaGrigliaDiPartenza()`. Il metodo `creaGrigliaDiPartenza()` istanzia quattro oggetti `Pilota` assegnando loro dei nomi. Poi istanzia quattro oggetti `Auto` assegnando loro i nomi delle scuderie ed i relativi piloti. Infine istanzia ed inizializza l'array griglia con le auto appena create. Una gara, dopo essere stata istanziata, è pronta per essere corsa.

Il metodo `corriGara()` contiene codice che va analizzato con più attenzione. Nella prima riga, infatti, viene chiamato il metodo `random()` della classe `Math` (appartenente al package `java.lang` che viene importato automaticamente). La classe `Math` astrae il concetto di *matematica* e sarà descritta più avanti in questo libro. Essa contiene metodi che astraggono classiche funzioni matematiche, come la radice quadrata o il logaritmo. Tra questi metodi utilizziamo il metodo `random()` che restituisce un numero generato in maniera casuale di tipo `double`, compreso tra 0 ed 0,9999999... (ovvero il numero `double` immediatamente più piccolo di 1). Nell'esercizio abbiamo moltiplicato per 4 questo numero, ottenendo un numero `double` casuale compreso tra 0 e 3,99999999... Questo è poi convertito ad intero, quindi vengono troncate tutte le cifre decimali. Abbiamo quindi ottenuto che la variabile `numeroVincente` immagazzini al runtime un numero generato casualmente, compreso tra 0 e 3, ovvero i possibili indici dell'array griglia.

Il metodo `corriGara()` genera quindi un numero casuale tra 0 e 3. Lo utilizza per individuare l'oggetto `Auto` dell'array griglia che vince la gara, per poi impostare il risultato tramite il metodo `dammiDettagli()` dell'oggetto `Auto` (che scriverà il lettore). Tutti gli altri metodi della classe sono di tipo accessor e mutator.

Esercizio 6.c) Modificatori d'accesso, static, e package, Vero o Falso:

1. Una classe dichiarata `private` non può essere utilizzata fuori dal package in cui è dichiarata.

2. La seguente dichiarazione di classe è scorretta:

```
public static class Classe {...}
```

3. La seguente dichiarazione di classe è scorretta:

```
protected class Classe {...}
```

4. La seguente dichiarazione di metodo è scorretta:

```
public void static metodo () {...}
```

5. Un metodo statico può utilizzare solo variabili statiche e, perché sia utilizzato, non bisogna per forza istanziare un oggetto dalla classe in cui è definito.

6. Se un metodo è dichiarato `static`, non può essere chiamato al di fuori del proprio package.

7. Una classe `static` non è accessibile fuori dal package in cui è dichiarata.

8. Un metodo `protected` viene ereditato in ogni sottoclasse qualsiasi sia il suo package.

9. Una variabile `static` viene condivisa da tutte le istanze della classe a cui appartiene.

10. Se non anteponiamo modificatori ad un metodo il metodo è accessibile solo all'interno dello stesso package.

Esercizio 6.d) Object Orientation in Java (pratica), Vero o Falso:

1. Un metodo statico deve essere per forza pubblico.

2. L'implementazione dell'incapsulamento implica l'utilizzo delle parole chiave `set` e `get`.

3. Per utilizzare le variabili incapsulate di una superclasse in una sottoclasse bisogna dichiararle almeno `protected`.

4. I metodi dichiarati privati non vengono ereditati nelle sottoclassi.

5. Un inizializzatore d'istanza viene invocato prima di ogni costruttore.

6. Una variabile privata risulta direttamente disponibile (tecnicamente come se fosse pubblica) tramite l'operatore dot, a tutte le istanze della classe in cui è dichiarata.

7. La parola chiave `this` permette di referenziare i membri di un oggetto che sarà creato solo al runtime all'interno dell'oggetto stesso.
8. La parola chiave `this` è sempre facoltativa.
9. La parola chiave `this` permette di chiamare un costruttore, da un metodo della stessa classe con la sintassi `this()`. È necessario però che questa sia la prima istruzione del metodo.
10. Il singleton pattern permette di creare una classe che può essere istanziata una sola volta.

Esercizio 6.e)

Astrarre il concetto di `Moneta` con una classe (completa di commenti). Assumiamo che tutte le monete avranno come valuta l'EURO, ed avranno la variabile d'istanza incapsulata `valore`. Ha senso creare una moneta senza specificarne il valore? Creare un vincolo affinché le monete possano essere istanziate obbligatoriamente con un valore.

Conviene stampare una frase in ogni metodo importante per potere verificare l'avvenuta esecuzione del nostro codice. Per esempio quando si istanzia una moneta. Questo vale anche per i prossimi esercizi.

Esercizio 6.f)

Considerando la classe `Moneta` creata nell'esercizio precedente, è corretto creare la variabile `valore` incapsulata con i metodi `setValore()` e `getValore()`? Modificare la classe in modo tale da astrarre al meglio la classe.

Esercizio 6.g)

Creare una classe `TestMonete` con un metodo `main()` che istanzia una moneta da 20 centesimi e una moneta da 1 centesimo ed eseguire l'applicazione. C'è qualcosa che non va su quanto viene stampato? Se è così, cambiare il codice in modo tale che le stampe siano senza errori grammaticali.

Esercizio 6.h)

Nella classe `TestMonete` istanziare anche una moneta da 1 euro. Probabilmente ci sarà un altro baco nella stampa, risolverlo. Aggiungere inoltre un metodo `getDescrizione()` che ritorna una stringa descrittiva della moneta corrente.

Esercizio 6.i)

Creare una classe (completa di commenti) `PortaMonete` che astrae il concetto di portamonete. Questo deve poter contenere al massimo 10 monete (la classe `Moneta` dovrebbe essere già stata creata nell'esercizio precedente). Per ora creare solo un costruttore che consenta di impostare il contenuto delle monete.

Anche in questo caso conviene stampare la descrizione delle azioni che si stanno invocando.

Esercizio 6.l)

Nella classe `TestMonete` istanziare anche un oggetto `PortaMonete` con 8 monete e uno con 11 monete, verificare che tutto funzioni correttamente. Aggiornare eventualmente anche il commento della classe.

Esercizio 6.m)

Creare un metodo `aggiungi()` all'interno della classe che permetta di aggiungere una moneta al portamonete. Prevedere gli appositi controlli di consistenza.

Esercizio 6.n)

Creare un metodo `stato()` che stampi l'attuale contenuto del portamonete.

Esercizio 6.o)

Creare un metodo `preleva()` all'interno della classe che consenta di prelevare (e quindi rimuovere) una moneta dal portamonete. Prevedere gli appositi controlli di consistenza.

Esercizio 6.p)

Modificare la classe `TestMonete` in modo tale da testare nella maniera più completa le classi create.

Esercizio 6.q)

Incapsulare la classe `Utente` dell'esercizio 5.z, e modificare di conseguenza la classe `Autenticazione` affinché tutto continui a funzionare.

Esercizio 6.r)

Disegnare un class diagram contente le due classi `Utente` e `Autenticazione` modificate nell'esercizio precedente che mostrano le loro variabili e i loro metodi. Consultare l'appendice L contenente uno schema di riferimento per la sintassi UML. In particolare per i membri statici (che vanno sottolineati) e la notazione di *aggregazione* che sussiste tra la classe contenitore `Autenticazione` e la classe contenuta `Utente`. Usare anche la notazione di molteplicità. Inoltre includere le due classi nella notazione di package.

Esercizio 6.s)

Abbiamo già notato alla fine dell'esercizio 5.z, che la soluzione di codice implementata non ci soddisfaceva. Infatti durante la nostra analisi avevamo individuato un flusso di esecuzione basato su classi e metodi differenti, come mostrato nei sequence diagram delle soluzioni dell'esercizio 5.v, che riportiamo per comodità.

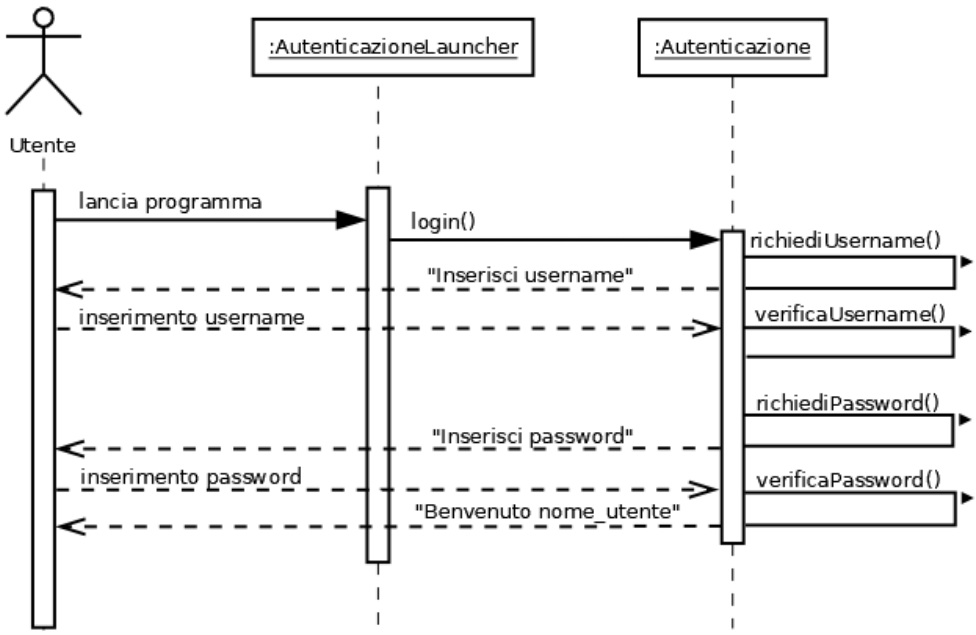

Figura 6.s.1 - (uguale alla 5.v.1) Sequence diagram che rappresenta lo scenario principale.

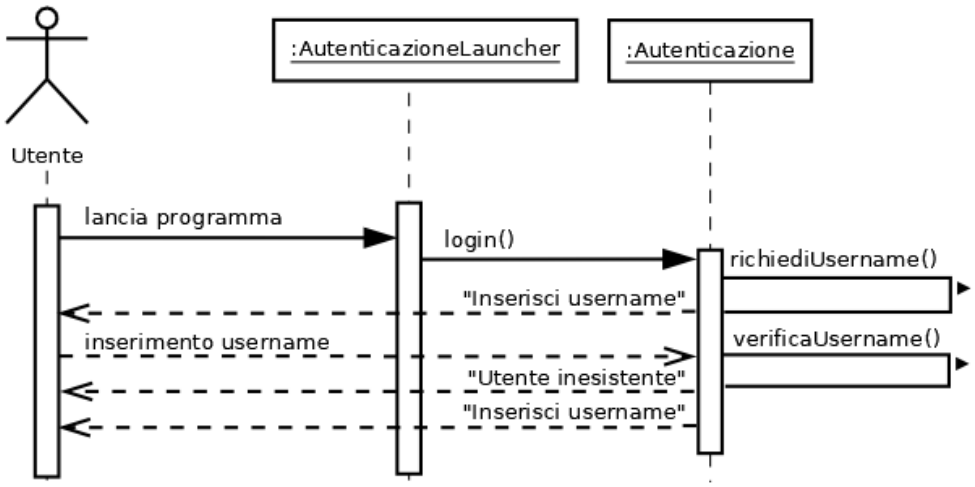

Figura 6.s.2 - (uguale alla 5.v.2) Sequence diagram che rappresenta il secondo scenario.

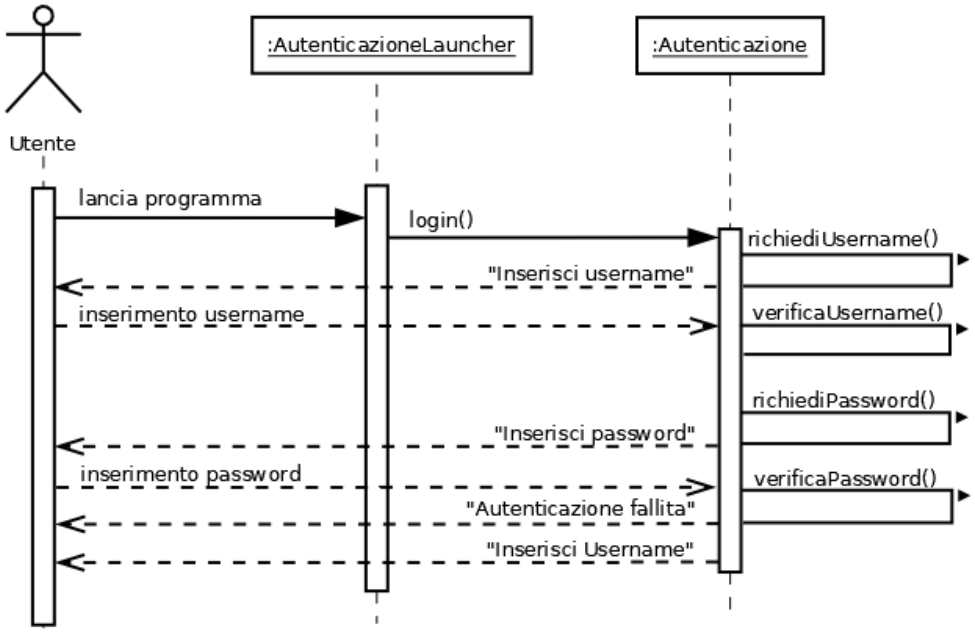

Figura 6.s.3 - (uguale alla 5.v.3) Sequence diagram che rappresenta il terzo scenario.

Creare un class diagram che rappresenti correttamente le classi necessarie a far funzionare gli scenari descritti con i sequence diagram, supportandosi con la syntax reference dell'appendice L. Inoltre trasformare la classe `Autenticazione` in un

singleton, come descritto nel paragrafo 6.8.6.
È possibile anche definire nuovi metodi o variabili se opportuno. Se non si riescono a definire i particolari della classe (per esempio tipi di ritorno, nomi o tipi di parametri e così via, semplicemente non definirli).

Ovviamente la classe `Utente` deve rimanere incapsulata.

Esercizio 6.t)

Una volta implementa la soluzione dell'esercizio precedente, dovremmo avere una classe `Autenticazione` con diversi metodi e responsabilità. Ha un unico metodo pubblico che gestisce il flusso (`login()`) e diversi metodi privati, alcuni verificano la correttezza dei dati inseriti dall'utente (utilizzando la variabile d'istanza `utenti`) ed altri per stampare messaggi all'utente. Un oggetto che si chiama `Autenticazione` è giusto che abbia la responsabilità di:

1. Gestire il flusso di esecuzione dell'applicazione.
2. Verificare la correttezza dei dati inseriti dall'utente.
3. Contenere la lista degli utenti.
4. Stampare messaggi.

Le responsabilità identificano l'astrazione della classe, e questa classe è sicuramente troppo carica di responsabilità. Facciamo quindi evolvere il nostro class diagram: cerchiamo di astrarre la classe `Autenticazione`, nella maniera più corretta e trovare altre astrazioni (classi) che possano implementare le responsabilità più specifiche. Partiamo dal trovare una classe che contenga i dati su cui vogliamo lavorare. Come la chiamiamo? Che variabili e metodi deve contenere? Siccome dobbiamo pensare in maniera object oriented, cerchiamo di renderla il più possibile riutilizzabile, ma anche coerente con il contesto in cui la stiamo definendo.

Esercizio 6.u)

Continuando l'esercizio precedente, modificare il class diagram individuando una classe che abbia le responsabilità di stampare i messaggi a video.

Esercizio 6.v)

Continuando l'esercizio precedente, la classe `Autenticazione` è ora astratta correttamente? Confermare o modificare ancora il diagramma.

Esercizio 6.z)

In base alla conclusione dell'esercizio precedente, implementare la soluzione di codice più vicina alla soluzione progettata. Rispetto alla soluzione a cui eravamo arrivati nell'esercizio 5.z, dovremmo avere la stessa funzionalità, ma un codice più semplice con cui interagire, astratto meglio, e più riutilizzabile.

Soluzioni degli esercizi del capitolo 6

Soluzione 6.a) Object Orientation in generale (teoria), Vero o Falso:

1. **Falso**, esiste dagli anni '60.
2. **Vero**.
3. **Falso**, ogni linguaggio fornisce supporto ai vari paradigmi in maniera diversa.
4. **Vero**.
5. **Vero**.
6. **Vero**.
7. **Vero**.
8. **Vero**.
9. **Vero**.
10. **Falso**, bisogna conoscere l'interfaccia pubblica e non l'implementazione interna.

Soluzione 6.b)

Il listato potrebbe essere simile al seguente:

```
public class Pilota {
    private String nome;
```

```
    public Pilota(String nome) {
        setNome(nome);
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }
}

public class Auto {
    private String scuderia;
    private Pilota pilota;

    public Auto(String scuderia, Pilota pilota) {
        setScuderia(scuderia);
        setPilota(pilota);
    }

    public void setScuderia(String scuderia) {
        this.scuderia = scuderia;
    }

    public String getScuderia() {
        return scuderia;
    }

    public void setPilota(Pilota pilota) {
        this.pilota = pilota;
    }

    public Pilota getPilota() {
        return pilota;
    }

    public String dammiDettagli() {
        return getPilota().getNome() + " su " + getScuderia();
    }
}
```

Soluzione 6.c) Modificatori e package, Vero o Falso:

1. **Falso**, `private` non si può utilizzare con la dichiarazione di una classe.
2. **Vero**, `static` non si può utilizzare con la dichiarazione di una classe.

3. **Vero**, `protected` non è applicabile a classi.
4. **Vero**, `static` deve essere posizionato prima del `void`.
5. **Vero**.
6. **Falso**, `static` non è un modificatore d'accesso.
7. **Falso**, `static` non è applicabile a classi.
8. **Vero**.
9. **Vero**.
10. **Vero**.

Soluzione 6.d) Object Orientation in Java (pratica), Vero o Falso:

1. **Falso**.
2. **Falso**, non si tratta di parole chiave ma solo di una convenzione.
3. **Falso**, possono essere private ed essere utilizzate tramite i metodi accessor e mutator.
4. **Vero**.
5. **Vero**.
6. **Vero**.
7. **Vero**.
8. **Falso**, nel caso ci sia ambiguità tra nomi di variabili d'istanza e locali, la parola chiave `this` è fondamentale (cfr. paragrafo 6.3.4).
9. **Falso**, solo un altro costruttore della stessa classe può usare quella sintassi.
10. **Vero**.

Soluzione 6.e)

Il listato potrebbe essere simile al seguente:

```
/**
 * Questa classe astrae il concetto di Moneta (esercizio 5.e)
 *
 * @author Claudio De Sio Cesari
 */
```

```
public class Moneta {

    /**
     * La valuta è una costante impostata al valore EURO.
     */
    public final static String VALUTA = "EURO";

    /**
     * Rappresenta il valore della moneta in centesimi.
     */
    private int valore;

    /**
     * Costruttore che prende in input il valore della moneta.
     *
     * @param valore il valore della moneta.
     */
    public Moneta(int valore) {
        this.valore = valore;
        System.out.println("Creata una moneta da " + valore + " centesimi");
    }

    /**
     * Imposta la variabile d'istanza valore.
     *
     * @param valore contiene il valore a cui deve essere impostato il
     *               valore della variabile d'istanza valore.
     */
    public void setValore(int valore) {
        this.valore = valore;
    }

    /**
     * Restituisce il valore della variabile d'istanza valore.
     *
     * @return
     *          il valore della variabile d'istanza valore.
     */
    public int getValore() {
        return valore;
    }
}
```

È sufficiente un costruttore per specificare il vincolo richiesto. Inoltre la valuta, essendo fissa per tutta le monete, è stata dichiarata come costante statica.

Soluzione 6.f)

Nell'attuale situazione, dove le specifiche anno richiesto solo di creare una classe che abbia il vincolo di essere istanziata sempre con un valore, la domanda potrebbe

risultare ambigua. Tuttavia non avendo altri vincoli espliciti, è ragionevole pensare di avere i vincoli che esistono nel mondo reale. Una moneta che ha valore specificato (supponiamo 5 centesimi) non potrà mai cambiare il proprio valore. Quindi sembra superfluo quanto meno il metodo `setValore()`. Quindi sarebbe corretto rimuoverlo. È anche consigliabile rendere la variabile `final` per rinforzare il concetto di immutabilità. Di seguito il listato modificato:

```
/**
 * Questa classe astrae il concetto di Moneta.
 *
 * @author Claudio De Sio Cesari
 */
public class Moneta {

    /**
     * La valuta è una costante impostata al valore EURO.
     */
    public final static String VALUTA = "EURO";

    /**
     * Rappresenta il valore della moneta in centesimi.
     */
    private final int valore;

    /**
     * Costruttore che prende in input il valore della moneta.
     *
     * @param valore il valore della moneta.
     */
    public Moneta(int valore) {
        this.valore = valore;
        System.out.println("Creata una moneta da " + valore
          + " centesimi di " + VALUTA);
    }

    /**
     * Restituisce il valore della variabile d'istanza valore.
     *
     * @return
     *          il valore della variabile d'istanza valore.
     */
    public int getValore() {
        return valore;
    }
}
```

Il codice è più compatto, ma forse almeno per l'inizio conviene utilizzare le variabili per meglio memorizzare le definizioni.

Soluzione 6.g)

Il listato della classe `TestMonete` potrebbe essere il seguente:

```
public class TestMonete {
    public static void main(String args[]) {
        Moneta monetaDaVentiCentesimi = new Moneta(20);
        Moneta monetaDaUnCentesimo = new Moneta(1);
    }
}
```

Eseguendo quest'applicazione l'output sarà:

```
Creata una moneta da 20 centesimi di EURO
Creata una moneta da 1 centesimi di EURO
```

Ma sarebbe più giusto che nella seconda riga la parola "centesimi" sia al singolare. Per risolvere questo problema potremmo modificare la classe `Moneta` nel seguente modo (riportiamo solo il costruttore responsabile della stampa e un metodo di utilità):

```
    public Moneta(int valore) {
        this.valore = valore;
        System.out.println("Creata una moneta da " +
          formattaUnitaDiMisura(valore) + VALUTA);
    }

    private static String formattaUnitaDiMisura(int valore) {
        return valore + (valore == 1 ?  " centesimo di " : " centesimi di ");
    }
```

Abbiamo delegato ad un nuovo metodo d'utilità privato la formattazione di un pezzo della frase da stampare usando un semplice operatore ternario (cfr. paragrafo 4.3.2), ed abbiamo risolto il nostro baco. Ora rieseguendo la classe `TestMonete` otterremo il seguente output:

```
Creata una moneta da 20 centesimi di EURO
Creata una moneta da 1 centesimo di EURO
```

Soluzione 6.h)

Il listato della classe `TestMonete` dovrebbe solo essere arricchito da un'istruzione di questo tipo:

```
Moneta monetaDaUnEuro = new Moneta(100);
```

L'esecuzione di questa applicazione genererà:

```
Creata una moneta da 20 centesimi di EURO
Creata una moneta da 1 centesimo di EURO
Creata una moneta da 100 centesimi di EURO
```

Ma sarebbe più giusto che nella terza riga "100 centesimi di EURO" fosse "1 EURO". Per risolvere questo problema potremmo modificare la classe `Moneta` nel seguente modo (riportiamo solo come cambiare il metodo di utilità):

```
    private static String formattaStringaDescrittiva (int valore) {
        String stringaFormattata = " centesimi di ";
        if (valore == 1) {
            stringaFormattata = " centesimo di ";
        } else if (valore > 99){
            stringaFormattata = " ";
            valore /= 100;
        }
        return valore + stringaFormattata;
    }
```

Abbiamo modificato il metodo d'utilità privato introdotto nell'esercizio precedente. Non è stato possibile usare l'operatore ternario, quindi abbiamo usato un costrutto `if`, ed abbiamo risolto il nostro baco. Ora rieseguendo la classe `TestMonete` otterremo il seguente output:

```
Creata una moneta da 20 centesimi di EURO
Creata una moneta da 1 centesimo di EURO
Creata una moneta da 1 EURO
```

Il metodo `getDescrizione()` quindi potrebbe essere codificato così:

```
    /**
     * Ritorna una descrizione della moneta corrente.
     *
     * @return
     *          una descrizione della moneta corrente.
     */
    public String getDescrizione() {
        String descrizione = "moneta da "
          + formattaStringaDescrittiva(valore) + VALUTA;
        return descrizione;
    }
```

E quindi anche il costruttore potrebbe riusare questo metodo nel seguente modo:

```
    public Moneta(int valore) {
        this.valore = valore;
        System.out.println("Creata una " + getDescrizione());
    }
```

Soluzione 6.i)

Il listato della classe `PortaMonete` potrebbe essere il seguente:

```
/**
 * Astrae il concetto di portamonete che può contenere un numero
 * limitato di monete.
 *
 * @author Claudio De Sio Cesari
 */
public class PortaMonete {

    /**
     * Un array che contiene un numero limitato di monete.
     */
    private final Moneta[] monete = new Moneta[10];

    /**
     * Crea un oggetto portamonete contenente monete i cui valori sono
     * specificati dal varargs valori.
     *
     * @param valori
     *          un varargs di valori di monete.
     */
    public PortaMonete(int... valori) {
        int numeroMonete = valori.length;
        for (int i = 0; i < numeroMonete; i++) {
            if (i >= 10) {
                System.out.println(
            "Sono state inserite solo le prime 10 monete!");
                break;
            }
            monete[i] = new Moneta(valori[i]);
        }
    }
}
```

Si noti che abbiamo usato un array di 10 oggetti `Moneta` dichiarato `final`, che farà da contenitore delle nostre monete. Inoltre abbiamo usato un varargs `valori`, per impostare il contenuto del `PortaMonete`. Questo ci sarà comodo quando effettivamente creeremo oggetti portamonete.
Nel caso siano passati più di dieci valori al costruttore, questo imposterà solo i primi dieci e stamperà un messaggio di avvertimento.

Soluzione 6.l)

Il listato della classe `TestMonete` modificato dovrebbe essere qualcosa di simile al seguente:

```
/**
 * Classe di test per le classi Moneta e PortaMonete.
 *
 * @author Claudio De Sio Cesari
 */
public class TestMonete {

    public static void main(String args[]) {
        Moneta monetaDaVentiCentesimi = new Moneta(20);
        Moneta monetaDaUnCentesimo = new Moneta(1);
        Moneta monetaDaUnEuro = new Moneta(100);
        // Creaiamo un portamonete con 8 monete
        PortaMonete portaMonete =
          new PortaMonete(2, 5, 100, 10, 50, 10, 100, 200);
        // Creaiamo un portamonete con 11 monete
        PortaMonete portaMoneteInsufficiente =
          new PortaMonete(2, 5, 100, 10, 50, 10, 100, 200, 10, 5, 2);
    }
}
```

L'output dovrebbe essere il seguente:

```
Creata una moneta da 20 centesimi di EURO
Creata una moneta da 1 centesimo di EURO
Creata una moneta da 1 EURO
Creata una moneta da 2 centesimi di EURO
Creata una moneta da 5 centesimi di EURO
Creata una moneta da 1 EURO
Creata una moneta da 10 centesimi di EURO
Creata una moneta da 50 centesimi di EURO
Creata una moneta da 10 centesimi di EURO
Creata una moneta da 1 EURO
Creata una moneta da 2 EURO
Creata una moneta da 2 centesimi di EURO
Creata una moneta da 5 centesimi di EURO
Creata una moneta da 1 EURO
Creata una moneta da 10 centesimi di EURO
Creata una moneta da 50 centesimi di EURO
Creata una moneta da 10 centesimi di EURO
Creata una moneta da 1 EURO
Creata una moneta da 2 EURO
Creata una moneta da 10 centesimi di EURO
Creata una moneta da 5 centesimi di EURO
Sono state inserite solo le prime 10 monete!
```

Soluzione 6.m)

Il codice risolutore proposto crea come soluzione anche un metodo privato d'utilità che restituisce il primo indice dell'array libero per contenere la nuova moneta:

```
    /**
     * Aggiunge una moneta al portamonete. Se questo è pieno la moneta
     * non è aggiunta e viene stampato un errore significativo.
     *
     * @param moneta
     *          la moneta da aggiungere.
     */
    public void aggiungi(Moneta moneta) {
        System.out.println("Proviamo ad aggiungere una " +
          moneta.getDescrizione());
        int indiceLibero = primoIndiceLibero();
        if (indiceLibero == -1) {
            System.out.println("Portamonete pieno! La moneta " +
              moneta.getDescrizione() + " non è stata aggiunta...");
        } else {
            monete[indiceLibero] = moneta;
            System.out.println("E' stata aggiunta una " +
              moneta.getDescrizione());
        }
    }

    /**
     * Restituisce il primo indice libero nell'array delle monete o -1
     * se il portamonete è pieno.
     *
     * @return
     *          il primo indice libero nell'array delle monete o -1
     *          se il portamonete è pieno.
     */
    private int primoIndiceLibero() {
        int indice = -1;
        for (int i = 0; i < 10; i++) {
            if (monete[i] == null) {
                indice = i;
                break;
            }
        }
        return indice;
    }
```

Soluzione 6.n)

Il listato per il metodo `stato()` potrebbe essere il seguente:

```
    /**
     * Stampa il contenuto del portamonete.
     */
    public void stato() {
        System.out.println("Il portamonete contiene:");
        for (Moneta moneta : monete) {
```

```
            if (moneta == null) {
                break;
            }
            System.out.println("Una " + moneta.getDescrizione());
        }
    }
```

Soluzione 6.o)

Il listato per il metodo `preleva()` potrebbe essere il seguente (anche in questo caso abbiamo creato un metodo privato di utilità):

```
    /**
     * Esegue un prelievo della moneta specificata dal portamonete
     * corrente. Nel caso non sia presente la moneta specificata, un
     * errore significativo verrà stampato e null verrà ritornato.
     *
     * @param moneta
     *          la moneta da prelevare.
     * @return
     *          la moneta trovata o null se non trovata.
     */
    public Moneta preleva(Moneta moneta) {
        System.out.println("Proviamo a prelevare una " +
          moneta.getDescrizione());
        Moneta monetaTrovata = null;
        int indiceMonetaTrovata = indiceMonetaTrovata(moneta);
        if (indiceMonetaTrovata == -1) {
            System.out.println("Moneta non trovata!");
        } else {
            monetaTrovata = moneta;
            monete[indiceMonetaTrovata] = null;
            System.out.println("Una " + moneta.getDescrizione()
              + " prelevata");
        }
        return monetaTrovata;
    }

    private int indiceMonetaTrovata(Moneta moneta) {
        int indiceMonetaTrovata = -1;
        for (int i = 0; i < 10; i++) {
            if (monete[i] == null) {
                break;
            }
            int valoreMonetaNelPortaMoneta = monete[i].getValore();
            int valore = moneta.getValore();
            if (valore == valoreMonetaNelPortaMoneta) {
                indiceMonetaTrovata = i;
                break;
            }
```

```
        }
        return indiceMonetaTrovata;
    }
```

Soluzione 6.p)

Come soluzione proponiamo un codice che cerca di testare anche le situazioni di errore:

```
/**
 * Classe di test per la classe Moneta.
 *
 * @author Claudio De Sio Cesari
 */
public class TestMonete {
    public static void main(String args[]) {
        Moneta monetaDaVentiCentesimi = new Moneta(20);
        Moneta monetaDaUnCentesimo = new Moneta(1);
        Moneta monetaDaUnEuro = new Moneta(100);
        // Creaiamo un portamonete con 11 monete
        PortaMonete portaMoneteInsufficiente =
          new PortaMonete(2, 5, 100, 10, 50, 10, 100, 200, 10, 5, 2);
        // Creaiamo un portamonete con 8 monete
        PortaMonete portaMonete =
          new PortaMonete(2, 5, 100, 10, 50, 10, 100, 200);
        portaMonete.stato();
        // Aggiungiamo una moneta da 20 centesimi
        portaMonete.aggiungi(monetaDaVentiCentesimi);
        // Aggiungiamo la decima moneta da 1 centesimo.
        portaMonete.aggiungi(monetaDaUnCentesimo);
        // Aggiungiamo l'undicesima moneta (dovremmo ottenere un
        // errore e la moneta non sarà aggiunta)
        portaMonete.aggiungi(monetaDaUnEuro);
        // Valutiamo lo stato del portamonete.
        portaMonete.stato();
        // preleviamo 20 centesimi
        portaMonete.preleva(monetaDaVentiCentesimi);
        // Aggiungiamo l'undicesima moneta (dovremmo ottenere un
        // errore e la moneta non sarà aggiunta)
        portaMonete.aggiungi(monetaDaUnEuro);
        portaMonete.stato();
        //Cerchiamo una moneta non esistente (dovremmo ottenere una
        // stampa di errore)
        portaMonete.preleva(new Moneta(7));
    }
}
```

L'output dovrebbe essere il seguente:

```
Creata una moneta da 20 centesimi di EURO
```

```
Creata una moneta da 1 centesimo di EURO
Creata una moneta da 1 EURO
Creata una moneta da 2 centesimi di EURO
Creata una moneta da 5 centesimi di EURO
Creata una moneta da 1 EURO
Creata una moneta da 10 centesimi di EURO
Creata una moneta da 50 centesimi di EURO
Creata una moneta da 10 centesimi di EURO
Creata una moneta da 1 EURO
Creata una moneta da 2 EURO
Creata una moneta da 10 centesimi di EURO
Creata una moneta da 5 centesimi di EURO
Sono state inserite solo le prime 10 monete!
Creata una moneta da 2 centesimi di EURO
Creata una moneta da 5 centesimi di EURO
Creata una moneta da 1 EURO
Creata una moneta da 10 centesimi di EURO
Creata una moneta da 50 centesimi di EURO
Creata una moneta da 10 centesimi di EURO
Creata una moneta da 1 EURO
Creata una moneta da 2 EURO
Il portamonete contiene:
Una moneta da 2 centesimi di EURO
Una moneta da 5 centesimi di EURO
Una moneta da 1 EURO
Una moneta da 10 centesimi di EURO
Una moneta da 50 centesimi di EURO
Una moneta da 10 centesimi di EURO
Una moneta da 1 EURO
Una moneta da 2 EURO
Proviamo ad aggiungere una moneta da 20 centesimi di EURO
E' stata aggiunta una moneta da 20 centesimi di EURO
Proviamo ad aggiungere una moneta da 1 centesimo di EURO
E' stata aggiunta una moneta da 1 centesimo di EURO
Proviamo ad aggiungere una moneta da 1 EURO
Portamonete pieno! La moneta moneta da 1 EURO non è stata aggiunta...
Il portamonete contiene:
Una moneta da 2 centesimi di EURO
Una moneta da 5 centesimi di EURO
Una moneta da 1 EURO
Una moneta da 10 centesimi di EURO
Una moneta da 50 centesimi di EURO
Una moneta da 10 centesimi di EURO
Una moneta da 1 EURO
Una moneta da 2 EURO
Una moneta da 20 centesimi di EURO
Una moneta da 1 centesimo di EURO
Proviamo a prelevare una moneta da 20 centesimi di EURO
Una moneta da 20 centesimi di EURO prelevata
Proviamo ad aggiungere una moneta da 1 EURO
E' stata aggiunta una moneta da 1 EURO
```

```
Il portamonete contiene:
Una moneta da 2 centesimi di EURO
Una moneta da 5 centesimi di EURO
Una moneta da 1 EURO
Una moneta da 10 centesimi di EURO
Una moneta da 50 centesimi di EURO
Una moneta da 10 centesimi di EURO
Una moneta da 1 EURO
Una moneta da 2 EURO
Una moneta da 1 EURO
Una moneta da 1 centesimo di EURO
Creata una moneta da 7 centesimi di EURO
Proviamo a prelevare una moneta da 7 centesimi di EURO
Moneta non trovata!
```

Soluzione 6.q)

Il listato di `Utente` incapsulato è il seguente:

```
package com.claudiodesio.autenticazione;

public class Utente {
    private String nome;
    private String username;
    private String password;

    public Utente(String n, String u, String p) {
        this.nome = n;
        this.username = u;
        this.password = p;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUsername() {
        return username;
```

```
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

mentre `Autenticazione` cambia poco: bisogna solo sostituire l'accesso diretto alle variabili pubbliche di `Utente`, con le relative chiamate ai metodi accessor:

```
package com.claudiodesio.autenticazione;

import java.util.Scanner;

public class Autenticazione {

    private static final Utente[] utenti = {
        new Utente("Daniele", "dansap", "musica"),
        new Utente("Giovanni", "giobat", "scienze"),
        new Utente("Ligeia", "ligder", "arte")
    };

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("Inserisci username.");
            String username = scanner.nextLine();
            Utente utente = verificaUsername(username);
            if (utente == null) {
                System.out.println("Utente non trovato!");
                continue;
            }
            System.out.println("Inserisci password");
            String password = scanner.nextLine();
            if (password != null &&
                password.equals(utente.getPassword())) {
                System.out.println("Benvenuto " + utente.getNome());
                break;
            } else {
                System.out.println("Autenticazione fallita");
            }
        }
    }

    private static Utente verificaUsername(String username) {
        if (username != null) {
            for (Utente utente : utenti) {
                if (username.equals(utente.getUsername())) {
                    return utente;
                }
            }
```

```
            }
            return null;
        }
    }
```

Soluzione 6.r)

La figura 6.r.1 mostra il class diagram richiesto. Si noti come i membri della classe `Autenticazione` siano marcati statici con una sottolineatura. Inoltre l'aggregazione (che indica la relazione di contenimento) viene indirizzata dall'oggetto contenuto all'oggetto contenente, e si distingue sintatticamente da una semplice associazione (relazione d'uso) tramite il disegno di un rombo bianco sul lato dell'oggetto contenuto. Il simbolo di asterisco `*` accanto all'oggetto contenuto invece, descrive la molteplicità dell'oggetto contenuto. Si noti infine che sul lato dell'oggetto contenente non ci sono simboli di molteplicità, questo significa che è come se fosse presente la molteplicità di default, ovvero `1`. Infatti un oggetto `Autenticazione` contiene più oggetti `Utente`.

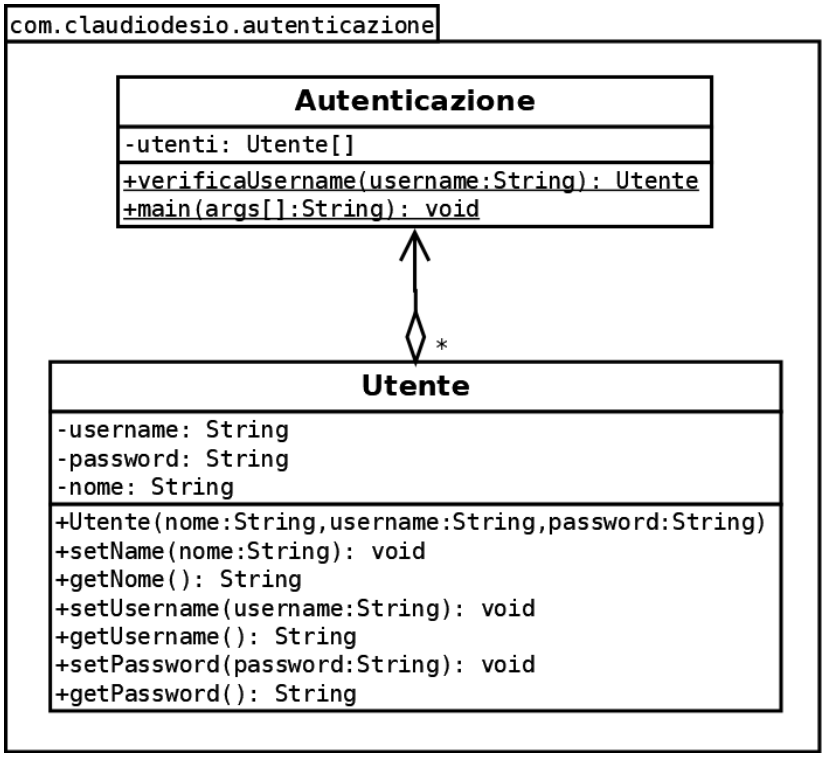

Figura 6.r.1 - Class diagram del package com.claudiodesio.autenticazione.

Soluzione 6.s)

La figura 6.s.1 mostra il class diagram richiesto. Notiamo che per trasformare la classe in singleton, abbiamo definito un costruttore privato, una variabile statica `instance` di tipo `Autenticazione`, e un metodo `getInstance()` che ha la responsabilità di restituire sempre la stessa istanza di `Autenticazione`, ovvero l'istanza `instance`, che viene istanziata opportunamente un'unica volta.

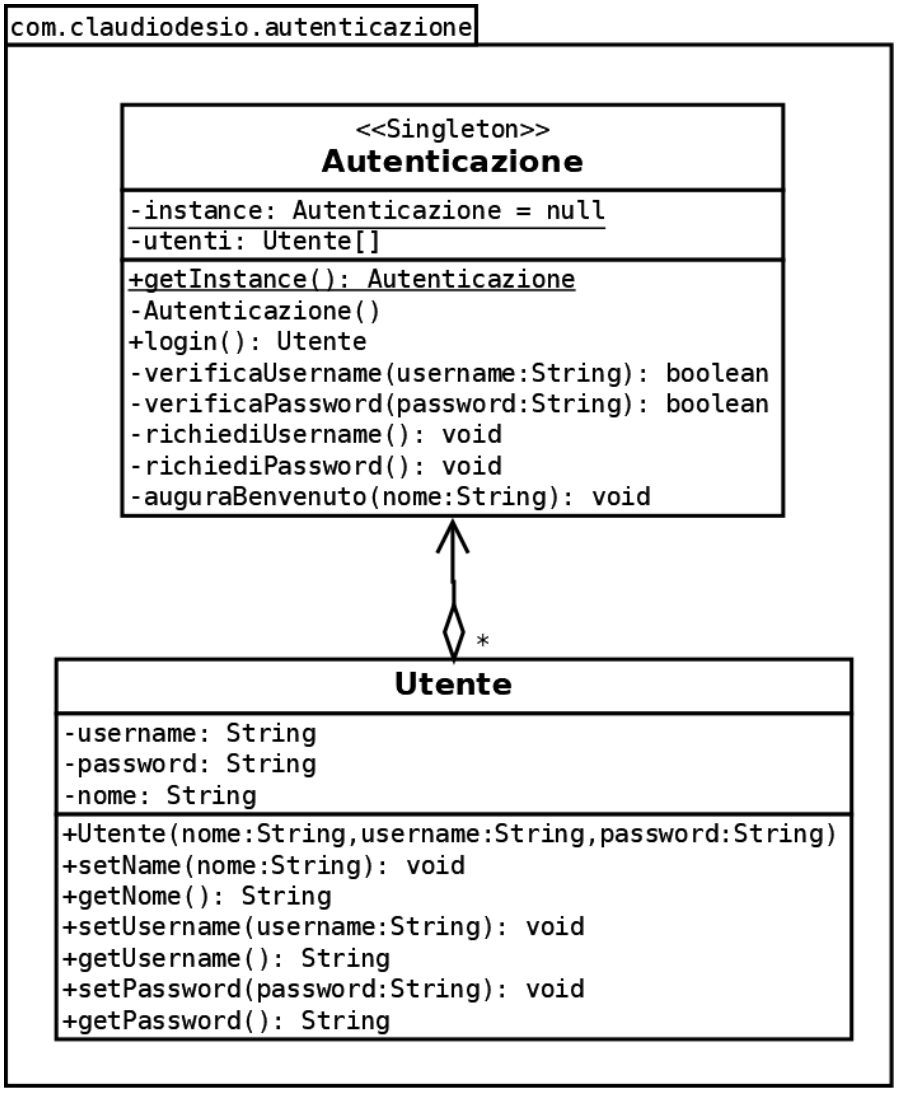

Figura 6.s.1 - Class diagram del package com.claudiodesio.autenticazione modificato come richiesto.

Visto che la classe è ora un singleton e verrà istanziata, abbiamo fatto in modo che i suoi metodi e le sue variabili non siano più statici. Abbiamo scritto tutti i metodi che sono stati descritti nei sequence diagram e abbiamo aggiunto i tipi di ritorno e i parametri a nostro avviso più corretti. In realtà scopriremo se questi sono davvero corretti quando affronteremo l'implementazione, questa per ora è solo la nostra idea, ed anche un'idea superficiale (si pensi alla soluzione di codice che abbiamo implementato nell'esercizio 5.z, dove il risultato non è quello a cui volevamo arrivare). In particolare abbiamo inteso i metodi `richiediUsername()` e `richiediPassword()` come metodi di stampa. Infatti non prendono in input parametri e nemmeno dichiarano tipi di ritorno. I metodi `verificaUsername()` e `verificaPassword()` invece dovrebbero tornare un booleano (`true` se la verifica va a buon fine e `false` se non va a buon fine). Abbiamo anche aggiunto i metodi `auguraBenvenuto()`, `autenticazioneFallita()`, e `usernameInesistente()` intesi come metodi di stampa, anche se non sono stati riportati nei sequence diagram (i nomi sono autoesplicativi). Tutti questi metodi sono metodi privati, mentre l'unico metodo pubblico è il metodo `login()`, che gestisce il flusso delle chiamate ai metodi privati.
Nella nostra mente la classe `Autenticazione` deve funzionare così (per ora!).

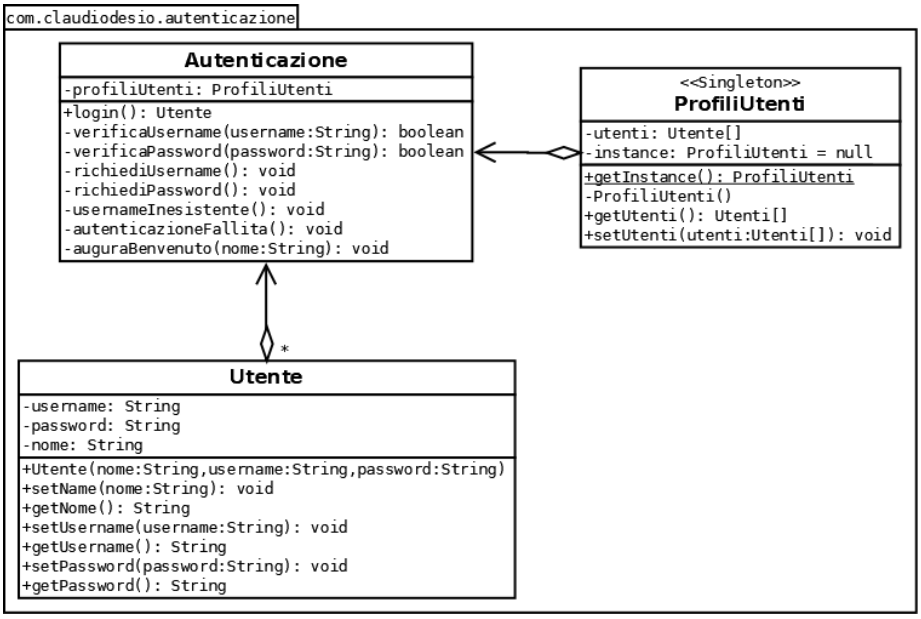

Figura 6.t.1 - Class diagram del package com.claudiodesio.autenticazione con ProfiliUtenti.

Soluzione 6.t)

Come è possibile verificare in figura 6.t.1, abbiamo creato una nuova classe chiamata `ProfiliUtenti`, che funge da base di dati e che contiene le informazioni sugli utenti (l'array `utenti`). Abbiamo deciso che un'istanza di questa classe sostituirà l'array `utenti` che prima risiedeva all'interno della classe `Autenticazione`, per non far perdere alla classe `Autenticazione` l'informazione sugli utenti. Adesso le classi sono astratte meglio, in quanto ognuna ha un ruolo specifico. Ci siamo resi conto che il design pattern singleton ha più senso che sia implementato nella classe `ProfiliUtenti`, rispetto che in `Autenticazione`. Infatti sono i dati che devono essere unici per tutte le classi, non il processo di autenticazione. Quindi abbiamo agito di conseguenza.

Soluzione 6.u)

Abbiamo creato una semplice classe di utilità chiamata `Stampa`, contenente tutti metodi che mandano messaggi all'utilizzatore dell'applicazione. Li abbiamo resi tutti statici, perché sembra superfluo istanziare una classe che contenga solo metodi di stampa senza che definisca variabili d'istanza.

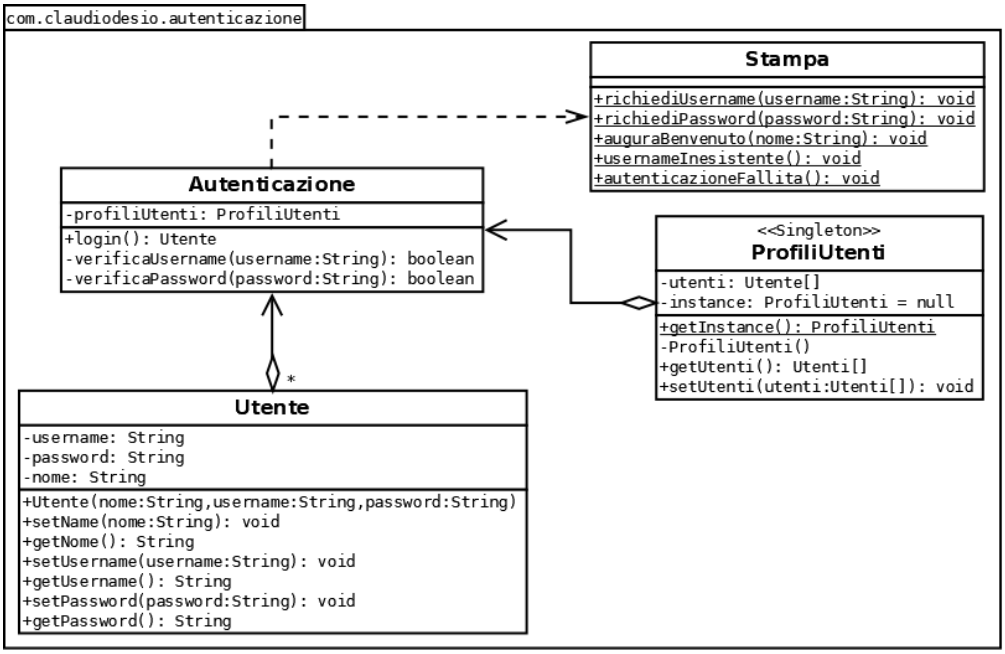

Figura 6.u.1 - Class diagram del package com.claudiodesio.autenticazione con Stampa.

Quest'ultima è una scelta come un'altra, non è detto che sia la migliore. Dichiarare statici i metodi implica ignorare i vantaggi dell'estensibilità che vedremo nei prossimi capitoli, ma la nostra scelta non è da condannare.

Soluzione 6.v)

La classe è astratta correttamente? Dipende dai punti di vista! La classe `Autenticazione`, ha come responsabilità la definizione del flusso della login, e la verifica della correttezza dei dati. Messa così sembra vada bene. Tuttavia si potrebbe anche pensare di delegare la verifica della correttezza dei dati ad un'altra classe d'utilità che potremmo chiamare `Verificatore`. Questa classe potrebbe contenere i due metodi di verifica dichiarati statici, oppure potrebbe contenere un costruttore a cui passiamo l'istanza di `Utente` che vogliamo verificare. Sono scelte tutte valide, ognuna delle quali ha delle conseguenze. Non creare la classe `Verificatore` implicherebbe avere una classe `Autenticazione` più corposa, ma crearla implicherebbe una classe in più (tra l'altro dipendente strettamente dalla classe `Utente`). Per ora optiamo per lasciare le cose come stanno. Decideremo più tardi quando avremo un quadro più chiaro.

Soluzione 6.z)

Come abbiamo detto, la classe `Utente` rimane invariata:

```
package com.claudiodesio.autenticazione;

public class Utente {

    private String nome;
    private String username;
    private String password;

    public Utente(String n, String u, String p) {
        this.nome = n;
        this.username = u;
        this.password = p;
    }

    public String getNome() {
        return nome;
    }
```

```
    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

La classe `ProfiliUtenti` è stata implementata in maniera fedele rispetto a come era stata progettata:

```
package com.claudiodesio.autenticazione;

public class ProfiliUtenti {

    private static ProfiliUtenti instance;

    private Utente[] utenti;

    private ProfiliUtenti() {
        utenti = creaUtenti();
    }

    public static ProfiliUtenti getInstance() {
        if (instance == null) {
            instance = new ProfiliUtenti();
        }
        return instance;
    }

    private Utente[] creaUtenti() {
        Utente[] utenti = {
            new Utente("Daniele", "dansap", "musica"),
            new Utente("Giovanni", "giobat", "scienze"),
            new Utente("Ligeia", "ligder", "arte")
        };
        return utenti;
    }
```

```
    public void setUtenti(Utente[] utenti) {
        this.setUtenti(utenti);
    }

    public Utente[] getUtenti() {
        return utenti;
    }
}
```

Anche la classe `Stampa` è fedele a come era stata progettata, a meno dell'introduzione di un metodo privato in più: `stampaMessaggio()`, che centralizza l'istruzione di stampa. Questo può servirci in futuro se vorremo modificare il modo in cui stampiamo un messaggio, perché dovremo farlo solo in quel metodo e non in tutti gli altri:

```
package com.claudiodesio.autenticazione;

public class Stampa {

    public static void richiediUsername() {
        stampaMessaggio("Inserisci username.");
    }

    public static void richiediPassword() {
        stampaMessaggio("Inserisci password.");
    }

    public static void auguraBenvenuto(String nome) {
        stampaMessaggio("Benvenuto " + nome);
    }

    public static void usernameInesistente() {
        stampaMessaggio("Utente non trovato!");
    }

    public static void autenticazioneFallita() {
        stampaMessaggio("Autenticazione fallita");
    }

    private static void stampaMessaggio(String messaggio) {
        System.out.println(messaggio);
    }
}
```

Invece la classe `Autenticazione` ha subito dei cambiamenti:

```
package com.claudiodesio.autenticazione;

import java.util.Scanner;
```

```
public class Autenticazione {

    public void login() {
        boolean autorizzato = false;
        Scanner scanner = new Scanner(System.in);
        do {
            Stampa.richiediUsername();
            String username = scanner.nextLine();
            Utente utente = trovaUtente(username);
            if (utente != null) {
                Stampa.richiediPassword();
                String password = scanner.nextLine();
                if (verificaPassword(utente, password)) {
                    Stampa.auguraBenvenuto(utente.getNome());
                    autorizzato = true;
                } else {
                    Stampa.autenticazioneFallita();
                }
            } else {
                Stampa.usernameInesistente();
            }
        } while (!autorizzato);
    }

    private Utente trovaUtente(String username) {
        Utente[] utenti = ProfiliUtenti.getInstance().getUtenti();
        if (username != null) {
            for (Utente utente : utenti) {
                if (username.equals(utente.getUsername())) {
                    return utente;
                }
            }
        }
        return null;
    }

//    private boolean verificaUsername(String username) {
//        Utente[] utenti = ProfiliUtenti.getInstance().getUtenti();
//        boolean trovato = false;
//        Utente utente = trovaUtente(username);
//        if (utente != null && username.equals(utente.getUsername())) {
//            trovato = true;
//        }
//        return trovato;
//    }

    private boolean verificaPassword(Utente utente, String password) {
        boolean trovato = false;
        if (password != null) {
            if (password.equals(utente.getPassword())) {
```

```
                    trovato = true;
                }
            }
            return trovato;
        }

        public static void main(String args[]) {
            Autenticazione autenticazione = new Autenticazione();
            autenticazione.login();
        }
    }
```

In particolare, il metodo `verificaUsername()`, che ritorna un booleano come avevamo progettato, è stato commentato e sostituito con il metodo `trovaUtente()`, che ritorna direttamente l'oggetto `Utente` relativo allo `username` specificato come parametro. Se non viene trovato nessun utente, il metodo ritorna `null`. Questa sostituzione ci permette di non duplicare codice (sia per trovare un utente con `trovaUtente()`, che per verificare lo username con `verificaUsername()`, avremmo effettuato lo stesso ciclo, e il codice dei due metodi sarebbe stato quasi identico).

Il metodo `main()` è stato introdotto solo come metodo per testare il funzionamento della funzionalità di login. Poteva esser posizionato in una classe qualsiasi, come per esempio `AutenticazioneLauncher` che avevamo inizialmente individuato nella nostra analisi tra le soluzioni degli esercizi del quinto capitolo.

Il metodo `login()` ora contiene la cosiddetta "logica di business", ovvero il codice che soddisfa i requisiti grazie ad un algoritmo opportuno. Rispetto alla soluzione dell'esercizio 5.z, si noti l'eliminazione dei costrutti `break` e `continue`, sostituiti dal più conveniente ciclo `do while`, supportato dalla variabile booleana `autorizzato`, che viene impostata a `true` solo nel momento in cui le procedure di verifica dello username e della password sono entrambe verificate. L'algoritmo risulta più chiaro e lineare, anche grazie al supporto della classe `Stampa` e dell'oggetto `Scanner`, che vengono utilizzati più volte rispettivamente per stampare messaggi in output e raccogliere input dell'utente dell'applicazione. Tuttavia a nostro parere, dei miglioramenti all'algoritmo e all'astrazione della classe si possono ancora fare. Infatti abbiamo dovuto improvvisare la nostra soluzione, perché quella progettata con il class diagram non si è rivelata meritevole di essere implementata.

Cosa è mancato?

È mancato ridisegnare gli scenari con gli interaction diagram, alla luce delle nuove classi e dei nuovi metodi individuati. In questi interaction diagram, avremmo potuto specificare anche i dettagli come tipi di parametri, nomi di oggetti e tipi di ritorno. Infatti i primi sequence diagram che abbiamo creato nell'esercizio 5.v,

erano basati solo sulle key abstraction, e servivano per verificare cosa fare (erano diagrammi di analisi). I diagrammi che avremmo potuto creare dopo le modifiche fatte al class diagram invece, avrebbero dovuto essere considerati diagrammi di progettazione che spiegassero come fare.
Per ora un passo in avanti grazie al class diagram lo abbiamo fatto, in seguito cercheremo di farne altri.

Esercizi del capitolo 7

Ereditarietà ed interfacce

Per questo capitolo eviteremo di far scrivere troppo codice al lettore. È molto importante invece concentrarsi piuttosto sulle definizioni. Se non si conoscono bene tutte i concetti della teoria, si finirà con lo scrivere codice incoerente dal punto di vista della filosofia ad oggetti.

Dopo ogni esercizio guardare la soluzione, perché ognuno potrebbe essere propedeutico al successivo.

Esercizio 7.a) Object Orientation in Java (teoria), Vero o Falso:

1. L'implementazione dell'ereditarietà implica scrivere sempre qualche riga in meno.
2. La seguente dichiarazione di classe è scorretta:

```
public final class Classe extends AltraClasse {...}
```

3. L'ereditarietà è utile solo se si utilizza la specializzazione. Infatti, specializzando ereditiamo nella sottoclasse (o sottoclassi) membri della superclasse che non bisogna riscrivere. Con la generalizzazione invece creiamo una classe in più, e quindi scriviamo più codice.

4. La parola chiave `super` permette di chiamare metodi e costruttori di superclassi. La parola chiave `this` consente di chiamare metodi e costruttori della stessa classe in cui ci si trova.
5. L'ereditarietà multipla non esiste in Java perché non esiste nella realtà.
6. Un'interfaccia funzionale è un'interfaccia che dichiara un unico metodo di default.
7. Una sottoclasse è più "grande" di una superclasse (nel senso che solitamente aggiunge caratteristiche e funzionalità nuove rispetto alla superclasse).
8. Supponiamo di sviluppare un'applicazione per gestire un torneo di calcio. Esiste ereditarietà derivata da specializzazione tra le classi `Squadra` e `Giocatore`.
9. Supponiamo di sviluppare un'applicazione per gestire un torneo di calcio. Esiste ereditarietà derivata da generalizzazione tra le classi `Squadra` e `Giocatore`.
10. In generale, se avessimo due classi `Padre` e `Figlio`, non esisterebbe ereditarietà tra queste due classi.

Esercizio 7.b)

Data la seguente classe:

```
public class Persona {
    private String nome;
    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }
}
```

Commentare la seguente classe `Impiegato` evidenziando dove sono utilizzati i paradigmi object oriented: incapsulamento, ereditarietà e riuso.

```
public class Impiegato extends Persona {

    private int matricola;

    public void setDati(String nome, int matricola) {
        setNome(nome);
```

```
        setMatricola(matricola);
    }

    public void setMatricola(int matricola) {
        this.matricola = matricola;
    }

    public int getMatricola() {
        return matricola;
    }

    public String dammiDettagli() {
        return getNome() + ", matricola: " + getMatricola();
    }
}
```

Esercizio 7.c) Classi astratte ed interfacce, Vero o Falso:

1. La seguente dichiarazione di classe è scorretta:

   ```
   public abstract final class Classe {...}
   ```

2. La seguente dichiarazione di classe è scorretta:

   ```
   public abstract class Classe;
   ```

3. La seguente dichiarazione di interfaccia è scorretta:

   ```
   public final interface Classe {...}
   ```

4. Una classe astratta contiene per forza metodi astratti.
5. Un'interfaccia può essere estesa da un'altra interfaccia.
6. Una classe può estendere una sola classe ma implementare più interfacce.
7. Il pregio delle classi astratte e delle interfacce è che obbligano le sottoclassi ad implementare i metodi astratti ereditati. Quindi rappresentano un ottimo strumento per la progettazione object oriented.
8. Un'interfaccia può dichiarare più costruttori.
9. Un'interfaccia non può dichiarare variabili ma costanti statiche e pubbliche.
10. Una classe astratta può implementare un'interfaccia.

Esercizio 7.d)

Descrivere tutte le relazioni di ereditarietà tra le seguenti classi:

1. `Professore`
2. `Studente`
3. `Persona`
4. `Cattedra`
5. `Corso`
6. `Aula`
7. `Lezione`

Esercizio 7.e)

Se volessimo creare la gerarchia definita nell'esercizio precedente, tra `Studente`, `Persona` e `Professore` quale potrebbe essere una classe astratta?

Esercizio 7.f)

Creare l'interfaccia (con commenti) `Musicale` che dichiara un metodo `suona()`. Come dichiarereste questo metodo: statico, di default o astratto?

Esercizio 7.g)

Creare due sottointerfacce (con commenti) di `Musicale`: `StrumentoMusicale` e `Suoneria`. Come dichiarereste il metodo `suona()` nelle due sottointerfacce: statico, di default o astratto?

Esercizio 7.h)

Supponiamo di creare una classe `Smartphone` che implementa entrambe le interfacce dell'esercizio precedente. Cosa c'è di sbagliato?

Esercizio 7.i) Interfacce in Java 8, Vero o Falso:

1. I metodi statici non si ereditano.
2. La seguente dichiarazione di interfaccia è scorretta:

```
public static interface Interface;
```

3. La seguente dichiarazione di interfaccia è scorretta:

```
public interface class {...}
```

4. I metodi astratti di un'interfaccia non vengono ereditati da un'altra interfaccia.

5. I metodi astratti di un'interfaccia non possono essere implementati da un'altra interfaccia.

6. I metodi statici di un'interfaccia non possono essere implementati da un'altra interfaccia.

7. Un'interfaccia `A`, definisce un metodo di default `m()`. L'interfaccia `B` estende l'interfaccia `A` ridefinendo il metodo `m()` con un'implementazione di default. Una classe astratta `C` implementa l'interfaccia `A` ma ridefinendo il metodo `m()`. Una classe concreta (non astratta) `D` estende la classe `C` e implementa l'interfaccia `B`, senza ridefinire il metodo `m()`. La classe `D` eredita il metodo `m()` definito nella classe `C`.

8. Un'interfaccia `E`, definisce un metodo astratto `m()`. L'interfaccia `F` estende l'interfaccia `E` ridefinendo il metodo `m()` con un'implementazione di default. Una classe astratta `G` implementa l'interfaccia `E` ma non ridefinisce il metodo `m()`. Una classe concreta (non astratta) `H` estende la classe astratta `G` e implementa l'interfaccia `F`, senza ridefinire il metodo `m()`. La classe `H` non può essere compilata correttamente.

9. Un'interfaccia può estendere più interfacce.

10. Un'interfaccia può estendere una classe astratta e un'interfaccia.

Esercizio 7.l)

Tenendo presente tutti gli inserimenti di codice che il compilatore esegue implicitamente, riscrivere la seguente classe, aggiungendo tutte le istruzioni che il compilatore aggiungerebbe:

```
public class CompilatorePensaciTu {

    private int var;

    public void setVar(int v) {
        var = v;
    }

    public int getVar() {
        return var;
    }
}
```

Esercizio 7.m)

Considerando le seguenti classi:

```
public class Persona {

    private String nome;

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return this.nome;
    }
}

public class Impiegato extends Persona {

    private int matricola;

    public void setMatricola(int matricola) {
        this.matricola = matricola;
    }

    public int getMatricola () {
        return this.matricola;
    }

    public String getDati() {
        return getNome() + "\nnumero" + getMatricola();
    }
}
```

Quale sarà l'output del processo di compilazione (scegliere una sola opzione)?

1. Nessun output (compilazione corretta).
2. Errore nel metodo `getDati()` di `Impiegato`.
3. Errore nel metodo `getNome()` di `Persona`.
4. Errore nel metodo `getMatricola()` di `Impiegato`.

Esercizio 7.n)

Se aggiungiamo alla classe `Impiegato` descritta nell'esercizio 7.m il seguente metodo:

```
public void setDati(String nome, int matricola) {
    setNome(nome);
    setMatricola(matricola);
}
```

Quale sarà l'output del processo di compilazione (scegliere una sola opzione)?

1. Nessun output (compilazione corretta).
2. Errore nel metodo `getDati()` di `Impiegato`.
3. Errore nel metodo `setNome()` di `Persona`.
4. Errore nel metodo `setMatricola()` di `Impiegato`.

Esercizio 7.o)

Quali di queste affermazioni è vera (potrebbero essere anche tutte vere)?

1. L'ereditarietà permette di mettere in relazione di aggregazione più classi.
2. L'ereditarietà permette di mettere in relazione di aggregazione più interfacce.
3. L'ereditarietà permette di mettere in relazione di estensione più classi ed interfacce.
4. L'ereditarietà permette di mettere in relazione di aggregazione più classi ed interfacce.

Esercizio 7.p)

Considerato il seguente codice:

```
class Animale {}
interface Felino {}
class Leone {}
```

Se volessimo mettere in relazione di ereditarietà i tipi precedenti, quali dei seguenti snippet è valido dal punto di vista del compilatore (potrebbero essere anche tutti validi)?

1. `class Animale extends Felino {}`
2. `interface Felino extends Animale {}`
3. `class Leone extends Felino {}`
4. `class Leone extends Animale implements Felino {}`
5. `class Animale extends Leone implements Felino {}`

Esercizio 7.q)

Dato che un'interfaccia può dichiarare oltre a metodi astratti, anche metodi statici pubblici e privati, e implementati (di default) pubblici e privati, e visto che con le interfacce possiamo implementare l'ereditarietà multipla, perché mai dovremmo preferire una classe astratta ad un'interfaccia?

Esercizio 7.r)

Dati i seguenti tipi:

- `interface Volante {}`
- `class Aereo implements Volante {}`

quali tra i seguenti snippet sono corretti?

1. `Aereo a = new Aereo();`
2. `Volante v = new Volante();`
3. `aereo1.equals(aereo2);` (dove `aereo1` e `aereo2` sono oggetti di tipo `Aereo`)
4. `Volante.aereo = new Aereo();`

Esercizio 7.s)

Quali modificatori sono implicitamente aggiunti a tutti i metodi di un'interfaccia (è possibile scegliere più di una risposta)?

1. `public`
2. `protected`
3. `private`
4. `static`
5. `default`
6. `abstract`
7. `final`

Esercizio 7.t)

Data la seguente gerarchia:

```
interface A {
    void metodo();
}

interface B extends A {}

abstract class C implements B {}

public final class D extends C {
    public void metodo() {}
}
```

Quali delle seguenti affermazioni sono false (è possibile scegliere più di un'affermazione):

1. La classe `C` non può essere dichiarata astratta perché implementa un'interfaccia, quindi questo codice non compila.
2. L'interfaccia `B` non può estendere un'altra interfaccia.
3. La classe `C` implementando `B`, eredita anche il metodo `metodo()` astratto di `A`.
4. La classe `D` non compila perché non può essere dichiarata `final`.
5. La classe `D` non compila perché è dichiarata `public`.
6. La classe `D` non compila perché il suo metodo è dichiarato `public`.

Esercizio 7.u)

Quali delle seguenti affermazioni sono vere (è possibile scegliere più di un'affermazione):

1. I metodi statici dichiarati in un'interfaccia non vengono ereditati nelle sottointerfacce.
2. Non è possibile dichiarare una classe `abstract` e `final` perché i due modificatori non sono compatibili tra loro.
3. Una classe astratta deve dichiarare per forza metodi astratti.
4. Le interfacce possono anche dichiarare costruttori.
5. Le classi astratte possono dichiarare metodi statici.

Esercizio 7.v)

Supponiamo di voler definire un tipo `Atleta`. Supponiamo che ogni atleta definisca i metodi `corri()` e `allenati()`. Supponiamo anche

di voler definire delle sottoclassi più specifiche come `Calciatore`, `Corridore` e `Tennista`. Come definireste `Atleta`, come interfaccia o classe astratta? Aggiungereste altri tipi?

La paternità di questo esercizio è da condividere con Raffaele Galiero, le cui acute osservazioni e il suo supporto hanno ispirato questo esercizio... grazie!

Esercizio 7.z)

Riprendiamo il caso di studio definito nel paragrafo 5.4. Avevamo fatto una serie di passi, che rappresentavano un possibile processo da seguire per poter creare infine del codice di qualità. In particolare avevamo definito i seguenti passi:

1. Analisi degli use case.
2. Definizione degli scenari per ogni use case.
3. Definire un high level deployment diagram di architettura.
4. Individuare le key abstraction.
5. Verificare la validità delle key abstraction utilizzando i diagrammi di iterazione per convalidare i flussi degli scenari, con gli oggetti istanziati dalla key abstraction.

Abbiamo visto negli esercizi finali relativi al capitolo 6, che altri passi da eseguire sono:

6. Definire le key abstraction su un class diagram.
7. Rivalutare il class diagram aggiungendo i dettagli essenziali e soprattutto ragionando sulle responsabilità.

Fare l'intero programma Logos è troppo impegnativo (ci vorrebbero settimane se non mesi di lavoro), ma possiamo concentrarci su particolari use case, e portare avanti il nostro processo solo su questi use case. Il lettore potrebbe poi anche iterare i ragionamenti fatti su ogni use case per completare pezzo dopo pezzo il programma. Anche se nell'analisi iniziale di Logos non è stato individuato il processo di autenticazione, in realtà ci deve essere! Infatti, dall'analisi degli use case erano stati definiti due attori: l'amministratore (che aveva compiti di configurazione del

sistema) e il commesso (che aveva compiti operativi). Sembra scontato che per potersi far riconoscere dal sistema, un meccanismo di login sia indispensabile. Quindi possiamo dire di aver scoperto un nuovo caso d'uso, che andiamo a definire come "autenticazione". Facciamo evolvere allora il diagramma dei casi d'uso della figura 5.4 nel diagramma di figura 7.v.1, dove introduciamo il nuovo caso d'uso trovato.

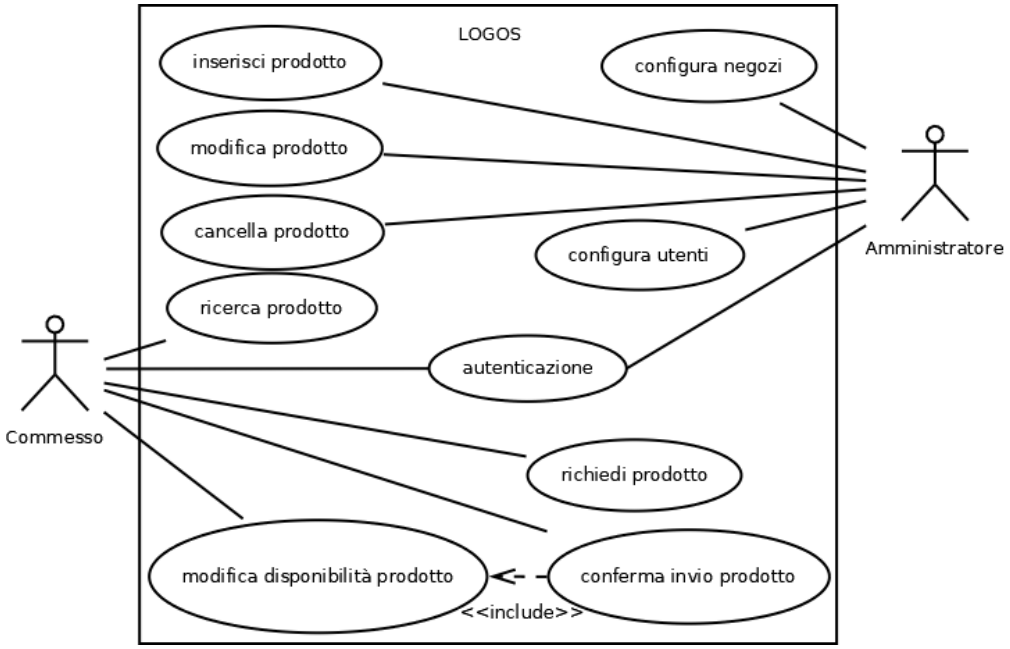

Figura 7.z.1 - Use case diagram aggiornato di Logos.

Per ora concentriamoci su questo caso d'uso calato nel contesto di Logos. Abbiamo il vantaggio di aver già lavorato su un programma che gestisce l'autenticazione con un certo flusso, vediamo se riusciamo ad evolverlo.
Quindi, tenendo presente che abbiamo già creato la classe `Utente`, vogliamo definire le classi `Commesso` e `Amministratore`. Conviene crearle? E perché?

Soluzioni degli esercizi del capitolo 7

Soluzione 7.a) Object Orientation in Java (pratica), Vero o Falso:

1. **Falso**, il processo di generalizzazione implica scrivere una classe in più e ciò non sempre implica scrivere qualche riga in meno.
2. **Falso**.
3. **Falso**, anche se dal punto di vista della programmazione la generalizzazione può non farci sempre risparmiare codice, essa ha comunque il pregio di farci gestire le classi in maniera più naturale, favorendo l'astrazione dei dati. Inoltre apre la strada all'implementazione del polimorfismo.
4. **Vero**.
5. **Falso**, l'ereditarietà multipla esiste nella realtà, e in Java esiste in una versione soft perché si eredita solo la parte funzionale.
6. **Falso**, è un'interfaccia che dichiara un unico metodo astratto.
7. **Vero**.
8. **Falso**, una squadra non "è un" giocatore, né un giocatore "è una" squadra. Semmai una squadra "ha un" giocatore ma questa non è la relazione di ereditarietà. Si tratta infatti della relazione di associazione.
9. **Vero**, infatti entrambe le classi potrebbero estendere una classe `Partecipante`.
10. **Falso**, un `Padre` è sempre un `Figlio`, o entrambe potrebbero estendere la classe `Persona`.

Soluzione 7.b)

```
public class Impiegato extends Persona { //Ereditarietà
    private int matricola;

    public void setDati(String nome, int matricola) {
        setNome(nome); //Riuso ed ereditarietà
        setMatricola(matricola); //Riuso
    }

    public void setMatricola(int matricola) {
        this.matricola = matricola; //incapsulamento
    }

    public int getMatricola() {
        return matricola; //incapsulamento
    }

    public String dammiDettagli() {
        //Riuso, incapsulamento ed ereditarietà
        return getNome() + ", matricola: " + getMatricola();
    }
}
```

Soluzione 7.c) Classi astratte ed interfacce, Vero o Falso:

1. **Vero**, i modificatori `abstract` e `final` sono in contraddizione.
2. **Vero**, manca il blocco di codice che definisce la classe.
3. **Vero**, un'interfaccia `final` non ha senso.
4. **Falso**.
5. **Vero**.
6. **Vero**.
7. **Vero**.
8. **Falso**, un'interfaccia non può dichiarare costruttori perché non si può istanziare.
9. **Vero**.
10. **Vero**.

Soluzione 7.d)

`Persona` potrebbe essere superclasse di `Professore` e `Studente`, per tutto il resto non c'è ereditarietà.

L'ereditarietà si testa sempre e comunque con la relazione "is a". Quindi risulta molto semplice verificare che per tutte le altre coppie di classi non è confermata questa relazione.

Soluzione 7.e)

Indubbiamente la classe `Persona` potrebbe essere una classe astratta, ma anche `Professore` e `Studente` potrebbero essere dichiarate astratte nel caso si volessero a loro volta estendere con classi come `StudenteIngegneria` o `ProfessoreMatematica`.

Soluzione 7.f)

Premettendo che tutte le scelte object oriented sono soggettive, probabilmente l'implementazione astratta è la scelta più corretta per un concetto così astratto:

```
/**
 * Astrae il concetto di oggetto musicale.
 *
 * @author Claudio De Sio Cesari
 */
public interface Musicale {
    /**
     * Esegue la musica dell'oggetto musicale corrente.
     */
    void suona();
}
```

Soluzione 7.g)

Il listato dell'interfaccia `StrumentoMusicale` potrebbe essere il seguente:

```
/**
 * Astrae il concetto di strumento musicale.
 *
 * @author Claudio De Sio Cesari
 */
public interface StrumentoMusicale extends Musicale {

}
```

Il listato dell'interfaccia `Suoneria` potrebbe essere il seguente:

```
/**
 * Astrae il concetto di suoneria musicale.
 *
 * @author Claudio De Sio Cesari
 */
public interface Suoneria extends Musicale {

}
```

Il metodo `suona()` per noi continua ad essere astratto.

Soluzione 7.h)

Mentre potrebbe essere plausibile considerare uno strumento musicale uno smartphone che usufruisce di una determinata app, è semplicemente scorretto che possa essere considerato una suoneria, infatti il test "is a" fallisce:

Domanda: "uno smartphone è uno strumento musicale?"
Risposta: Sì (a patto che sia installata un'app che lo renda tale)
Domanda: "uno smartphone è una suoneria?"
Risposta: No (semmai contiene suonerie)

Soluzione 7.i) Classi astratte ed interfacce, Vero o Falso:

1. **Vero**.
2. **Vero**, il modificatore `static` non si può applicare a classi ed interfacce.
3. **Vero**, `class` è una parola chiave e non può essere utilizzata come identificatore.
4. **Falso**.
5. **Falso**.
6. **Vero**, in particolare è possibile riscrivere un metodo con la stessa firma in una sottointerfaccia (ma lo stesso concetto si applica alle classi), ma tecnicamente non si tratta di un override perché i metodi statici semplicemente non vengono ereditati. Infatti un'eventuale uso nella sottointerfaccia dell'annotazione `@Override` per marcare il nuovo metodo statico provocherà un errore in compilazione (cfr. listati SuperInterfaccia.java e SottoInterfaccia.java).
7. **Vero**, la regola 5 che abbiamo visto nel paragrafo 7.7.5.5 sull'ereditarietà multipla e che abbiamo caratterizzato dalla massima "class always win", continua

a valere anche se la classe interessata è astratta (cfr. listati A.java, B.java, C.java, D.java, TestABCD.java).

8. **Falso**, potrebbe trarre in inganno il fatto che la classe astratta `G` anche ereditando il metodo astratto dall'interfaccia `E` faccia valere i motivi della regola 5 "class always win" studiata nel paragrafo 7.7.5.5. Invece in questo caso, vale la regola 4, ovvero vince l'implementazione più specifica. Infatti la classe `G`, eredita un metodo dell'interfaccia `E`, che è meno specifico di quello ridefinito nell'interfaccia `F` (cfr. listati E.java, F.java, G.java, H.java, TestEFGH.java).
9. **Vero**, anche se potrebbe sorgere qualche dubbio, un'interfaccia può estendere più interfacce (cfr. listato MiaInterfaccia.java).
10. **Falso**, un'interfaccia non può estendere una classe in qualsiasi caso.

Soluzione 7.l)

Il compilatore in realtà trasformerà la classe in qualcosa di molto simile alla seguente (in grassetto gli inserimenti impliciti del compilatore):

```
import java.lang.*;

public class CompilatorePensaciTu extends Object {

    private int var;

    public CompilatorePensaciTu() {
    }

    public void setVar(int v) {
      this.var = v;
    }

    public int getVar() {
      return this.var;
    }
}
```

Teniamo anche conto che, estendendo `Object`, questa classe eredita anche tutti i suoi metodi.

Soluzione 7.m)

La risposta esatta è la prima. Nessun errore da segnalare.

Soluzione 7.n)

La risposta esatta è la prima. Nessun errore da segnalare.

Soluzione 7.o)

Il concetto di aggregazione, utilizzato più volte negli esercizi precedenti, è una relazione che indica contenimento, che semmai può essere considerata un'alternativa all'estensione. Quindi l'unica risposta corretta è la terza.

Soluzione 7.p)

La quarta e la quinta opzione sono entrambe corrette per il compilatore. La quinta però, ha meno senso dal punto di vista della logica (un animale è un leone? Non per forza!).

Soluzione 7.q)

Anche se un'interfaccia può potenzialmente definire metodi di diversi tipi, non può dichiarare variabili d'istanza, ma solo costanti statiche pubbliche.

Soluzione 7.r)

Gli statement corretti sono i numeri 1 e 3.

> **Il metodo `equals()` è ereditato direttamente dalla classe `Object`.**

Soluzione 7.s)

Solo la prima risposta è corretta. Essendo la prima corretta, ovviamente la seconda e la terza non possono essere corrette. È possibile dichiarare metodi statici e di default, ma i relativi modificatori non sono mai aggiunti automaticamente. Il dubbio può venire per la risposta 6, perché prima dell'avvento di Java 8 questa risposta sarebbe stata giusta, ora non più visto che possiamo dichiarare nelle interfacce anche metodi di default e statici. Infine, il modificatore `final` è implicitamente aggiunto agli attributi delle interfacce (che sono anche implicitamente dichiarati statici e pubblici).

Soluzione 7.t)

Le risposte sono tutte false tranne la 3.

Soluzione 7.u)

Le risposte corrette sono la 1, la 2 e la 5.

Soluzione 7.v)

Potremmo seguire il seguente ragionamento. Un tennista si allena e corre diversamente da un calciatore o da un corridore. Insomma, sia la classe `Tennista`, sia la classe `Corridore` che la classe `Calciatore`, ridefiniranno i metodi `allenati()` e `corri()`. In `Atleta`, questi due metodi invece li vorremmo definire astratti, perché, a priori, sarebbe difficile definire come si allena o come corre un atleta... dipende dal tipo di atleta! A questo punto si potrebbe pensare di definire `Atleta` come interfaccia visto che utilizza solo due metodi astratti, e lo si può fare. Ma con l'evoluzione di questo programma, è altamente probabile che si vogliano definire campi di anagrafica degli atleti come potrebbero essere `nome` e `cognome`. Un'interfaccia però non può dichiarare variabili, e quindi si potrebbe preferire dichiarare `Atleta` come classe astratta. Oppure potremmo creare una soluzione dove `Tennista`, `Calciatore` e `Corridore` implementino l'interfaccia `Atleta` (che definisce i due metodi astratti `allenati()` e `corri()`) ed estendono la classe astratta `Persona` (che definisce gli attributi `nome`, `cognome`, etc.). Insomma sarà il contesto del programma in cui ci caleremo che ci porterà a definire la soluzione più corretta.

Soluzione 7.z)

Una soluzione semplice e quasi automatica, consisterebbe nell'estendere direttamente la classe `Utente`, con le due sottoclassi `Amministratore` e `Commesso`:

```
package com.claudiodesio.autenticazione;

public class Amministratore extends Utente {
    public Amministratore(String nome, String username, String password) {
        super(nome, username, password);
    }
}
```

e

```
package com.claudiodesio.autenticazione;

public class Commesso extends Utente {
    public Commesso(String nome, String username, String password) {
        super(nome, username, password);
    }
}
```

Solo che non abbiamo ancora informazioni sufficienti per inserire campi e metodi specifici per queste due classi (infatti sono vuote). Quindi per ora decidiamo di non implementarle.

Esercizi del capitolo 8

Polimorfismo

Per questo capitolo simuleremo la costruzione di un IDE di programmazione in maniera molto semplificata. Anche in questo caso sarà un esercizio incrementale quindi bisogna completare ogni esercizio (e leggerne la soluzione) per poter passare al successivo.
Creeremo classi ed interfacce andando per gradi. In particolare creeremo le astrazioni di `IDE`, `Editor`, `FileSorgente`, `File`, `TipoFile` e così via. L'esercizio è guidato, quindi il lettore è sollevato dalla responsabilità di decidere quali classi devono comporre la nostra applicazione. Per tale ragione si tratta di esercizi di programmazione più che di analisi e progettazione.
Con i prossimi esercizi ci sarà solo da applicare le definizioni che abbiamo imparato sino ad ora, non è richiesto quasi nessun tipo di algoritmo. Lo scopo finale è di concentrarsi sull'Object Orientation e non sugli algoritmi. Inoltre sono presentati anche altri tipi di esercizi come quelli che supportano la preparazione alla certificazione Oracle.

Esercizio 8.a) Polimorfismo per metodi, Vero o Falso:

1. L'overload di un metodo implica scrivere un altro metodo con lo stesso nome e diverso tipo di ritorno.
2. L'overload di un metodo implica scrivere un altro metodo con nome differente e stessa lista di parametri.
3. La segnatura (o firma) di un metodo è costituita dalla coppia identificatore - lista di parametri.

4. Per sfruttare l'override bisogna che sussista l'ereditarietà.
5. Per sfruttare l'overload bisogna che sussista l'ereditarietà.
6. Supponiamo che in una classe `B`, la quale estende la classe `A`, ereditiamo il metodo:

   ```
   public int m(int a, String b) { ... }
   ```

   Se nella classe B scriviamo il metodo:

   ```
   public int m(int c, String b) { ... }
   ```

   stiamo facendo overload e non override.
7. Se nella classe `B` scriviamo il metodo:

   ```
   public int m(String a, String b) { ... }
   ```

   stiamo facendo overload e non override.
8. Se nella classe `B` scriviamo il metodo:

   ```
   public void m(int a, String b) { ... }
   ```

   otterremo un errore in compilazione.
9. Se nella classe `B` scriviamo il metodo:

   ```
   protected int m(int a, String b) { ... }
   ```

   otterremo un errore in compilazione.
10. Se nella classe `B` scriviamo il metodo:

    ```
    public int m(String a, int c) { ... }
    ```

    otterremo un override.

Esercizio 8.b) Polimorfismo per dati, Vero o Falso:

1. Considerando le classi introdotte in questo capitolo, il seguente frammento di codice non produrrà errori in compilazione:

   ```
   Veicolo v [] = {new Automobile(), new Aereo(), new Aereo()};
   ```

2. Considerando le classi introdotte in questo capitolo, il seguente frammento di codice non produrrà errori in compilazione:

   ```
   Object o [] = {new Automobile(), new Aereo(), "ciao"};
   ```

3. Considerando le classi introdotte in questo capitolo, il seguente frammento di codice non produrrà errori in compilazione:

```
Aereo a [] = {new Veicolo(), new Aereo(), new Aereo()};
```

4. Considerando le classi introdotte in questo capitolo, e se il metodo della classe viaggiatore fosse questo:

```
public void viaggia(Object o) {
   o.accelera();
}
```

potremmo passargli un oggetto di tipo `Veicolo` senza avere errori in compilazione. Per esempio:

```
claudio.viaggia(new Veicolo());
```

5. Considerando le classi introdotte in questo capitolo, il seguente frammento di codice non produrrà errori in compilazione:

```
PuntoTridimensionale ogg = new Punto();
```

6. Considerando le classi introdotte in questo capitolo, il seguente frammento di codice non produrrà errori in compilazione:

```
PuntoTridimensionale ogg = (PuntoTridimensionale)new Punto();
```

7. Considerando le classi introdotte in questo capitolo, il seguente frammento di codice non produrrà errori in compilazione:

```
Punto ogg = new PuntoTridimensionale();
```

8. Considerando le classi introdotte in questo capitolo, e se la classe `Piper` estende la classe `Aereo`, il seguente frammento di codice non produrrà errori in compilazione:

```
Veicolo a = new Piper();
```

9. Considerando le classi introdotte in questo capitolo, il seguente frammento di codice non produrrà errori in compilazione:

```
String stringa = fiat500.toString();
```

10. Considerando le classi introdotte in questo capitolo. Il seguente frammento di codice non produrrà errori in compilazione:

```
public void pagaDipendente(Dipendente dip) {
  if (dip instanceof Dipendente) {
    dip.stipendio = 1000;
  }
  else if (dip instanceof Programmatore) {
    ...
  }
```

Esercizio 8.c)

Per iniziare, creare un'interfaccia `TipoFile` che definisca delle costanti statiche che rappresentano i tipi di file sorgente che vorremo gestire con l'IDE. Di sicuro una di queste costanti deve chiamarsi `JAVA`, tutte le altre sono a piacere. Scegliere anche il tipo delle costanti a piacere.

Esercizio 8.d)

Creare una classe `File` che astrae il concetto di file generico e che definisce un nome e un tipo.

Esercizio 8.e)

Creare una classe `FileSorgente` che astrae il concetto di file sorgente (estendendo la classe `File`) che definisce anche un contenuto di tipo stringa.

Esercizio 8.f)

Aggiungere un metodo `aggiungiTesto()` che aggiunga una stringa di testo alla fine del contenuto del file sorgente.

Esercizio 8.g)

Aggiungere un overload del metodo `aggiungiTesto()` che aggiunga una stringa di testo in un punto specificato del contenuto del file sorgente (consultare la documentazione della classe `String`).

Esercizio 8.h)

Creare una classe `TestFileSorgente` che testi il funzionamento corretto della classe `FileSorgente`.

Esercizio 8.i)

Creare un'interfaccia `Editor` che astrae il concetto di editor di testo. Bisogna definire metodi per aprire, chiudere, salvare e modificare un file.

Esercizio 8.l)

Creare un'interfaccia `IDE` che astrae il concetto di IDE di sviluppo. Si tenga presente che un IDE è anche un editor. Bisogna definire metodi per compilare ed eseguire un file.

Esercizio 8.m)

Creare una semplice implementazione `JavaIDE` della classe `IDE`. Aggiungere un'implementazione per il metodo `modifica()` (e a piacere è possibile reimplementare tutti i metodi che si ritiene utile).

Esercizio 8.n)

Creare una classe di test `TestIDE` che esegue delle operazioni sul file tramite IDE.

L'esercizio potrebbe poi continuare estendendo ulteriormente queste classi, sentitevi liberi di compiere altre iterazioni di programmazione dopo aver completato tutto. Dovrete darvi delle specifiche, capire come implementarle e implementarle: tre passi.

Esercizio 8.o) Varargs, Vero o Falso:

1. I varargs permettono di utilizzare i metodi come se fossero degli overload.
2. La seguente dichiarazione è compilabile correttamente:

```
public void myMethod(String... s, Date d) {
  ...
}
```

3. La seguente dichiarazione è compilabile correttamente:

```
public void myMethod(String... s, Date d...) {
  ...
}
```

4. Considerando il seguente metodo:

```
public void myMethod(Object... o) {
  ...
}
```

   la seguente invocazione è corretta:

```
oggetto.myMethod();
```

5. La seguente dichiarazione è compilabile correttamente:

```
public void myMethod(Object o, Object os...) {
  ...
}
```

6. Considerando il seguente metodo:

```
public void myMethod(int i, int... is) {
  ...
}
```

la seguente invocazione è corretta:

```
oggetto.myMethod(new Integer(1));
```

7. Le regole dell'override cambiano con l'introduzione dei varargs.

8. Il metodo di `java.io.PrintStream printf()` è basato sul metodo `format()` della classe `java.util.Formatter`.

9. Il metodo `format()` di `java.util.Formatter` non ha overload perché definito con un varargs.

10. Nel caso in cui si passi un array come varargs al metodo `printf()` di `java.io.PrintStream`, questo verrà trattato non come oggetto singolo ma come se fossero stati passati ad uno ad uno ogni suo elemento.

Esercizio 8.p)

Data la seguente gerarchia:

```
interface A {
    void metodo();
}

interface B implements A {
    static void metodoStatico() {}
}

final class C implements B {}

public abstract class D implements A {
    @Override
    void metodo() {}
}
```

Scegliere tutte le affermazioni vere:

1. L'interfaccia `B` non eredita il metodo `metodoStatico()`.
2. L'interfaccia `B` non può implementare un'altra interfaccia.
3. La classe `C` implementando `B` eredita anche il metodo `metodo()` astratto di `A`, e non essendo dichiarata astratta non può essere compilata.
4. La classe `D` non compila perché non può essere dichiarata `abstract`. Infatti non dichiara alcun metodo astratto.
5. La classe `D` non compila perché il metodo che dichiara non è dichiarato `public`.
6. La classe `D` compila solo perché il metodo è annotato con `Override`.

Esercizio 8.q)

Quali delle seguenti affermazioni è corretta (scegliere tutte quelle corrette):

1. Un'interfaccia estende la classe `Object`.
2. Un metodo che prende come parametro un reference di tipo `Object`, può prendere in input qualsiasi oggetto di qualsiasi tipo, anche di tipo interfaccia.
3. Un metodo che prende come parametro un reference di tipo `Object`, può prendere in input qualsiasi oggetto di qualsiasi tipo, anche un array.
4. Un metodo che prende come parametro un reference di tipo `Object`, può prendere in input qualsiasi oggetto di qualsiasi tipo, anche una collezione eterogenea.
5. Tutti i cast di oggetti, sono valutati al tempo di compilazione.

Esercizio 8.r)

Considerando le seguenti classi:

```
public class StampaNumero {
    public void stampa(double numero) {
        System.out.print(numero);
    }
}

public class StampaIntero extends StampaNumero {
    public void stampa(int numero) {
        System.out.print(numero);
    }
```

```
    public static void main(String args[]) {
        StampaNumero stampaNumero = new StampaIntero();
        stampaNumero.stampa(1);
    }
}
```

Eseguendo `StampaIntero`, qual è l'output di questo programma?

1. 1.2
2. 1
3. 1.0
4. 11.2

Esercizio 8.s)

Considerate la seguente gerarchia:

```
public interface Satellite {
    void orbita();
}

public class Luna implements Satellite {
    @Override
    public void orbita() {
        System.out.println("Luna che orbita");
    }
}

public class SatelliteArtificiale implements Satellite {
    @Override
    public void orbita() {
        System.out.println("Satellite artificiale che orbita");
    }
}
```

E la seguente classe di test:

```
public class TestSatelliti {
    public static void main(String args[]) {
        test(new Luna(), new SatelliteArtificiale());
        Satellite[] satelliti = {
            new Luna(), new SatelliteArtificiale()
        };
        test(satelliti);
        test();
        test(new Object());
```

```
    }

    public static void test(Satellite... satelliti) {
        for (Satellite satellite : satelliti) {
            satellite.orbita();
        }
    }
}
```

Scegliere tutte le affermazioni corrette:

1. L'applicazione compila e viene eseguita senza errori.
2. L'applicazione non compila per l'istruzione `test(satelliti);`
3. L'applicazione non compila per l'istruzione `test();`
4. L'applicazione non compila per l'istruzione `test(new Object());`
5. L'applicazione compila ma al runtime si blocca per un'eccezione.

Esercizio 8.t)

Definire l'overload e l'override. E fare un esempio di una sottoclasse, che implementa entrambi i concetti.

Esercizio 8.u)

Tenendo presente che `Number` è superclasse della classe `Integer`, consideriamo la seguente gerarchia:

```
public abstract class SommaNumero {
    public abstract Number somma (Number n1, Number n2);
}

public class SommaIntero extends SommaNumero {
    @Override
    public Integer somma(Number n1, Number n2) {
        return (Integer)n1 + (Integer)n2;
    }
}
```

Scegliere tutte le affermazioni corrette:

1. La classe `SommaIntero` compila senza errori.
2. La classe `SommaIntero` non compila perché l'override non è corretto: i tipi di ritorno non coincidono.

3. La classe `SommaIntero` non compila perché non è possibile utilizzare l'operatore `+` se non con numeri di tipo primitivo.
4. La classe `SommaIntero` potrebbe causare un'eccezione al runtime.

Esercizio 8.v)

Rendere robusto il metodo `somma()` della classe `SommaIntero` definito nell'esercizio 8.u, in modo che al runtime funzioni senza generare eccezioni.

Esercizio 8.z)

Definire brevemente cos'è un parametro polimorfo, cosa sono le collezioni eterogenee, e cosa è una chiamata virtuale ad un metodo.

Soluzioni degli esercizi del capitolo 8

Soluzione 8.a) Polimorfismo per metodi, Vero o Falso:

1. **Falso**, l'overload di un metodo implica scrivere un altro metodo con lo stesso nome e diversa lista di parametri.
2. **Falso**, l'overload di un metodo implica scrivere un altro metodo con lo stesso nome e diversa lista di parametri.
3. **Vero**.
4. **Vero**.
5. **Falso**, l'overload di un metodo implica scrivere un altro metodo con lo stesso nome e diversa lista di parametri.
6. **Falso**, stiamo facendo override. L'unica differenza sta nel nome dell'identificatore di un parametro, che è ininfluente al fine di distinguere metodi.
7. **Vero**, la lista dei parametri dei due metodi è diversa.
8. **Vero**, in caso di override il tipo di ritorno non può essere differente.
9. **Vero**, in caso di override il metodo riscritto non può essere meno accessibile del metodo originale.
10. **Falso**, otterremo un overload. Infatti, le due liste di parametri differiscono per posizioni.

Soluzione 8.b) Polimorfismo per dati, Vero o Falso:

1. **Vero**.
2. **Vero**.
3. **Falso**, la classe `Veicolo` essendo astratta non è istanziabile. Inoltre non è possibile inserire in una collezione eterogenea di aerei un `Veicolo` che è superclasse di `Aereo`.
4. **Falso**, la compilazione fallirebbe già dal momento in cui provassimo a compilare il metodo `viaggia()`. Infatti non è possibile chiamare il metodo `accelera()` con un reference di tipo `Object`.
5. **Falso**, c'è bisogno di un casting, perché il compilatore non sa a priori il tipo a cui punterà il reference al runtime.
6. **Vero**.
7. **Vero**.
8. **Vero**, infatti `Veicolo` è superclasse di `Piper`.
9. **Vero**, il metodo `toString()` appartiene a tutte le classi perché ereditato dalla superclasse `Object`.
10. **Vero**, ma tutti i dipendenti saranno pagati allo stesso modo.

Soluzione 8.c)

Il listato potrebbe essere il seguente:

```
public interface TipoFile {
    int JAVA = 1;
    int C_SHARP = 2;
    int C_PLUS_PLUS = 3;
    int C = 4;
}
```

Si noti che non è necessario specificare modificatori per le costanti dato che è sottointeso che siano dichiarate implicitamente `public`, `static` e `final`. Inoltre abbiamo scelto come tipo delle costanti `int`, ma un qualsiasi altro tipo sarebbe andato bene, l'importante è che le costanti abbiano valori differenti.

Questo tipo di uso di interfacce è in disuso da anni, precisamente da quando in Java versione 5 sono state introdotte ...

. . . le cosiddette enumerazioni (cfr. capitolo 10). Tuttavia useremo ugualmente questa modalità di programmazione visto che non abbiamo ancora affrontato l'argomento enumerazioni.

Soluzione 8.d)

Il listato della classe `File` potrebbe essere il seguente:

```
public abstract class File {

    private String nome;

    private int tipo;

    public File(String nome, int tipo) {
        this.nome = nome;
        this.tipo = tipo;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public int getTipo() {
        return tipo;
    }

    public void setTipo(int tipo) {
        this.tipo = tipo;
    }
}
```

Si noti che abbiamo dichiarato la classe astratta, perché generica e creata ai fini dell'estensione.

Soluzione 8.e)

Il listato della classe `FileSorgente` dovrebbe essere il seguente:

```
public class FileSorgente extends File {

    private String contenuto;
```

```
    public FileSorgente(String nome, int tipo) {
        super(nome, tipo);
    }

    public FileSorgente(String nome, int tipo, String contenuto) {
        this(nome, tipo);
        this.contenuto = contenuto;
    }

    public String getContenuto() {
        return contenuto;
    }

    public void setContenuto(String contenuto) {
        this.contenuto = contenuto;
    }
}
```

Si noti che abbiamo risfruttato il costruttore della superclasse mediante la parola chiave `super`, ed abbiamo creato anche un costruttore equivalente a quello della superclasse.

Soluzione 8.f)

Il listato del metodo richiesto potrebbe essere:

```
    public void aggiungiTesto(String testo) {
        if (contenuto == null) {
            contenuto = "";
        }
        if (testo != null) {
            contenuto += testo;
        }
    }
```

Una stringa `null` "sommata" ad un'altra stringa viene rappresentata proprio con la stringa "null". Da quest'affermazione si può intuire del perché abbiamo implementato il metodo in questo modo.

Soluzione 8.g)

Il listato del metodo richiesto potrebbe essere:

```
    public void aggiungiTesto(String testo, int posizione) {
        final int length = contenuto.length();
        if (contenuto != null && testo != null && posizione > 0
          && posizione < length) {
            contenuto = contenuto.substring(0, posizione) + testo +
```

```
                contenuto.substring(posizione);
            }
        }
```

Abbiamo scelto, per semplicità, di non aggiungere testo nel caso la posizione specificata non sia corretta. Tuttavia una clausola `else` che stampi un messaggio di errore potrebbe rappresentare una soluzione migliore. In realtà questo metodo si può migliorare molto... provateci!

Nel capitolo 9 vedremo come si gestiscono veramente le eccezioni in Java.

Soluzione 8.h)

Il listato per la classe `TestFileSorgente` potrebbe essere il seguente:

```
public class TestFileSorgente {
    public static void main(String args[]) {
        FileSorgente fileSorgente = new FileSorgente("Test.java",
          TipoFile.JAVA, "public class MyClass {\n\r");
        System.out.println(fileSorgente.getContenuto());
        // Test aggiungiTesto (String) corretto
        fileSorgente.aggiungiTesto("}");
        System.out.println(fileSorgente.getContenuto());
        // Test aggiungiTesto (String,int) corretto
        fileSorgente.aggiungiTesto("//Test aggiunta testo\n\r", 23);
        System.out.println(fileSorgente.getContenuto());
        // Test aggiungiTesto (String,int) scorretto
        fileSorgente.aggiungiTesto("//Test aggiunta testo\n\r", -1);
        System.out.println(fileSorgente.getContenuto());
        // Test aggiungiTesto (String,int) scorretto
        fileSorgente.aggiungiTesto("//Test aggiunta testo\n\r", 100);
        System.out.println(fileSorgente.getContenuto());
    }
}
```

L'output risultante è il seguente:

```
public class MyClass {

public class MyClass {
}
public class MyClass {
//Test aggiunta testo
}
public class MyClass {
```

```
//Test aggiunta testo
}
public class MyClass {
//Test aggiunta testo
}
```

Si potrebbero scrivere molti più casi di test, e si dovrebbe anche fare in modo che ogni caso di test non dipenda dal precedente, ma per ora va bene così.

Per effettuare casi di test con maggiore comodità, è consigliato l'utilizzo di un tool come JUnit. Potete trovare una breve descrizione di JUnit all'interno dell'appendice G nei paragrafi G.4.1 e G.4.2.

Soluzione 8.i)

Il listato dell'interfaccia richiesta potrebbe essere:

```
public interface Editor {
    default void salva(FileSorgente file) {
        System.out.println("File: " + file.getNome() + " salvato!");
    }
    default void apri(FileSorgente file) {
        System.out.println("File: " + file.getNome() + " aperto!");
    }
    default void chiudi(FileSorgente file) {
        System.out.println("File: " + file.getNome() + " chiuso!");
    }
    default void modifica(FileSorgente file, String testo) {
        System.out.println("File: " + file.getNome() + " modificato!");
    }
}
```

Abbiamo creato dei metodi di default che simulano con la stampa la loro effettiva esecuzione.

Soluzione 8.l)

Il listato dell'interfaccia richiesta potrebbe essere:

```
public interface IDE extends Editor {
    default void compila(FileSorgente file) {
        System.out.println("File: " + file.getNome() + " compilato!");
    }
```

```
    default void esegui(FileSorgente file) {
        System.out.println("File: " + file.getNome() + " eseguito!");
    }
}
```

Anche in questo caso abbiamo creato dei metodi di default che simulano con la stampa la loro effettiva esecuzione.

Soluzione 8.m)

Il listato della classe richiesta potrebbe essere:

```
public class JavaIDE implements IDE {
    @Override
    public void modifica(FileSorgente file, String testo) {
        IDE.super.modifica(file, testo);
        file.aggiungiTesto(testo);
        System.out.println("Contenuto modificato:\n" + file.getContenuto());
    }
}
```

Soluzione 8.n)

Il listato della classe richiesta potrebbe essere:

```
public class TestIDE {
    public static void main(String args[]) {
        IDE ide = new JavaIDE();
        FileSorgente fileSorgente = new FileSorgente("Test.java",
          TipoFile.JAVA, "public class MyClass {\n\r");
        ide.modifica(fileSorgente, "}");
    }
}
```

L'output risultante sarà:

```
File: Test.java modificato!
Contenuto modificato:
public class MyClass {
}
```

Soluzione 8.o) Varargs, Vero o Falso:

1. **Vero**.
2. **Falso**.
3. **Falso**.

4. **Vero**.
5. **Vero**.
6. **Vero**.
7. **Falso**.
8. **Vero**.
9. **Falso**.
10. **Vero**.

Soluzione 8.p)

Le risposte vere sono la 1, la 2, la 3 e la 5. La numero 5 è vera perché il metodo ereditato è implicitamente pubblico, e ridefinendolo senza il modificatore `public`, si sta cercando di renderlo meno accessibile di quello ereditato. Infatti compilando la classe `D` otterremo il seguente errore:

```
error: metodo() in D cannot implement metodo() in A
  void metodo() {}
       ^
  attempting to assign weaker access privileges; was public
1 error
```

Soluzione 8.q)

Le affermazioni corrette sono la 2, la 3 e la 4.

Soluzione 8.r)

La risposta giusta è la numero 3. Infatti, viene chiamato sempre il metodo `stampa()` della superclasse `StampaNumero` che prende in input un `double`, e questo spiega il formato dell'output. Il motivo per cui non viene chiamato in maniera virtuale il metodo nella sottoclasse `StampaIntero`, è perché non si tratta di un override, visto che il tipo di parametro è diverso tra i due metodi. Non trattandosi di un override, usando un reference della superclasse, l'unico metodo che si può chiamare, è proprio quello della superclasse.

Soluzione 8.s)

L'unica risposta corretta è la numero 4.

Soluzione 8.t)

Overload: dal momento che un metodo è univocamente determinato dalla sua firma, in una classe (o un'interfaccia) è possibile creare più metodi con lo stesso identificatore ma con differente lista di parametri. In casi come questo si parla di overload di un (nome di un) metodo.
Override: permette di riscrivere in una sottoclasse un metodo ereditato da una superclasse (o interfaccia). Un esempio di sottoclasse, che implementa entrambi i concetti, lo possiamo ricavare modificando le classi dell'esercizio 8.r:

```
public class StampaNumero {
    public void stampa(double numero) {
        System.out.print(numero);
    }
}

public class StampaIntero extends StampaNumero {
    //oveload
    public void stampa(int numero) {
        System.out.print(numero);
    }

    //overload e override
    public void stampa(double numero) {
        System.out.print(numero);
    }
}
```

Soluzione 8.u)

Le affermazioni corrette sono le numero 1 e 4.
L'affermazione 2 non è corretta perché il tipo di ritorno della classe `SommaIntero` è covariante (cfr. paragrafo 8.2.3). L'affermazione 3 invece non è corretta perché sussiste l'autoboxing-unboxing, di cui abbiamo già parlato nei paragrafi 3.3.2 e 4.3.4, e che approfondiremo nel paragrafo 11.1.2.
La numero 4 è corretta perché se per esempio eseguissimo questo codice:

```
SommaIntero sommaIntero = new SommaIntero();
sommaIntero.somma(1.0, 1.0);
```

otterremo quest'eccezione al runtime:

```
Exception in thread "main" java.lang.ClassCastException:
  java.base/java.lang.Double cannot be cast to
  java.base/java.lang.Integer
      at SommaIntero.somma(SommaIntero.java:4)
      at SommaIntero.main(SommaIntero.java:9)
```

Alla gestione delle eccezioni è dedicato gran parte del capitolo 9.

Soluzione 8.v)

Una possibile soluzione potrebbe essere la seguente:

```
public class SommaIntero extends SommaNumero {
    @Override
    public Integer somma(Number n1, Number n2) {
        if (n1 == null || n2 == null) {
            System.out.println("Impossibile sommare un operando null, "
              + "restituisco il valore di default");
            return Integer.MIN_VALUE;
        } else if (!(n1 instanceof Integer && n2 instanceof Integer)) {
            System.out.println("Passa solo variabili di tipo intere, "
              + "restituisco il valore di default");
            return Integer.MIN_VALUE;
        }
        return (Integer)n1 + (Integer)n2;
    }
}
```

Eseguendo il seguente metodo `main()` infatti:

```
    public static void main(String args[]) {
        SommaIntero sommaIntero = new SommaIntero();
        sommaIntero.somma(1.0, 1.0);
        sommaIntero.somma(null, 1.0);
    }
```

otterremo il seguente output e nessuna eccezione:

```
Passa solo variabili di tipo intere, restituisco il valore di default
Impossibile sommare un operando null, restituisco il valore di default
```

Si noti che il primo controllo sulla nullità dei parametri è necessario perché non è possibile utilizzare il cast su una variabile nulla, e nemmeno utilizzare l'operatore `+`.

Soluzione 8.z)

Un **parametro polimorfo** è un parametro di un metodo dichiarato di un certo tipo (magari astratto) ma che al runtime punterà ad un'istanza di una sua sottoclasse.

Le **collezioni eterogenee** sono collezioni di oggetti diversi, come per esempio un array di `Number`, che contiene oggetti delle sue sottoclassi come `Integer`.
Si parla di **chiamata virtuale ad un metodo** quando, utilizzando un reference di una superclasse, viene invocato un metodo che in realtà è ridefinito in una sottoclasse.

Esercizi del capitolo 9

Eccezioni ed asserzioni

La gestione delle eccezioni è un argomento fondamentale, è molto importante imparare ogni dettaglio dell'argomento (non sono molti in fondo). Le asserzioni sono molto meno usate, ma potrebbero essere usate con profitto.

Esercizio 9.a) Gestione delle eccezioni e degli errori, Vero o Falso:

1. Ogni eccezione che estende una `ArithmeticException` è una unchecked exception.
2. Un `Error` si differenzia da una `Exception` perché non può essere lanciato; infatti non estende la classe `Throwable`.
3. Il seguente frammento di codice:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (ArithmeticException exc) {
    System.out.println("Divisione per zero...");
}
catch (NullPointerException exc) {
    System.out.println("Reference nullo...");
}
catch (Exception exc) {
    System.out.println("Eccezione generica...");
}
```

```
finally {
    System.out.println("Finally!");
}
```

produrrà il seguente output:

```
Divisione per zero...
Eccezione generica...
Finally!
```

4. Il seguente frammento di codice:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (Exception exc) {
    System.out.println("Eccezione generica...");
}
catch (ArithmeticException exc) {
    System.out.println("Divisione per zero...");
}
catch (NullPointerException exc) {
    System.out.println("Reference nullo...");
}
finally {
    System.out.println("Finally!");
}
```

produrrà un'eccezione al runtime.

5. La parola chiave `throw` permette di lanciare "a mano" solo le sottoclassi di `Exception` che crea il programmatore.

6. La parola chiave `throw` consente di lanciare "a mano" solo le sottoclassi di `Exception`.

7. Se un metodo fa uso della parola chiave `throw`, affinché la compilazione abbia buon esito nello stesso metodo deve essere gestita l'eccezione che si vuole lanciare, o il metodo stesso deve utilizzare una clausola `throws`.

8. Non è possibile estendere la classe `Error`.

9. Se un metodo `m2` fa ovverride di un altro metodo `m2` posto nella superclasse, non potrà dichiarare con la clausola `throws` eccezioni nuove che non siano sottoclassi rispetto a quelle che dichiara il metodo `m2`.

10. Dalla versione 1.4 di Java è possibile "includere" in un'eccezione un'altra eccezione.

Esercizio 9.b) Gestione delle asserzioni, Vero o Falso:

1. Se in un'applicazione un'asserzione non viene verificata, si deve parlare di bug.

2. Un'asserzione che non viene verificata provoca il lancio da parte della JVM di un `AssertionError`.

3. Le precondizioni servono per testare la correttezza dei parametri di metodi pubblici.

4. È sconsigliato l'utilizzo di asserzioni laddove si vuole testare la correttezza di dati inseriti da un utente.

5. Una postcondizione serve per verificare che al termine di un metodo sia verificata un'asserzione.

6. Un'invariante interna permette di testare la correttezza dei flussi all'interno dei metodi.

7. Un'invariante di classe è una particolare invariante interna che deve essere verificata per tutte le istanze di una certa classe, in ogni momento del loro ciclo di vita, tranne che durante l'esecuzione di alcuni metodi.

8. Un'invariante sul flusso di esecuzione, è solitamente un'asserzione con una sintassi del tipo: `assert false;`

9. Non è in alcun modo possibile compilare un programma che fa uso di asserzioni con il jdk 1.3.

10. Non è in alcun modo possibile eseguire un programma che fa uso di asserzioni con il jdk 1.3.

Esercizio 9.c)

Consideriamo le classi create negli esercizi del capitolo 8: `File`, `FileSorgente`, `Editor`, `IDE` e `JavaIDE`. Consideriamo inoltre il metodo `aggiungiTesto(String)` che abbiamo creato nell'esercizio 8.f, che avevamo codificato nel seguente modo:

```
    public void aggiungiTesto(String testo) {
```

```
        if (contenuto != null && testo != null) {
            contenuto += testo;
        }
    }
```

Il controllo della clausola `if` si può sicuramente migliorare. Cosa usereste in questo caso, asserzioni o eccezioni?

Esercizio 9.d)

Dopo aver letto la soluzione dell'esercizio precedente, usare le parole chiave `throw` ed eventualmente `throws` per gestire eventuali eccezioni nel metodo `aggiungiTesto(String)` citato nell'esercizio precedente.

Esercizio 9.e)

Consideriamo ora il metodo `aggiungiTesto(String, int)` che abbiamo creato nell'esercizio 8.g e che avevamo codificato nel seguente modo:

```
    public void aggiungiTesto(String testo, int posizione) {
        final int length = contenuto.length();
        if (contenuto != null && testo != null && posizione > 0
          && posizione < length) {
            contenuto = contenuto.substring(0, posizione) + testo +
              contenuto.substring(posizione);
        }
    }
```

Gestire un'eccezione (o più eccezioni) tramite le parole chiave `try-catch`, e se possibile anche `finally`.

Esercizio 9.f)

Dopo aver svolto l'esercizio precedente, creare una classe di test `TestFileSorgente` per verificare che la gestione delle eccezioni funzioni correttamente.

Esercizio 9.g)

Consideriamo i sorgenti creati con gli esercizi realizzati per il capitolo 6. Creare delle eccezioni personalizzate per gestire le situazioni impreviste. In particolare creare un'eccezione che scatti nel costruttore della classe `PortaMonete` quando vengono specificate troppe monete, chiamiamola `PortaMonetePienoException`. Gestire l'eccezione direttamente nel costruttore istanziando comunque l'oggetto con un numero limitato di elementi (ovvero senza

cambiare il comportamento già definito nell'esercizio del capitolo 6). Gestire anche la `NullPointerException` (quale istruzione potrebbe generare tale eccezione?).

Esercizio 9.h)

Consideriamo i sorgenti creati con gli esercizi realizzati per il capitolo 6. Usare la gestione delle eccezioni personalizzate per gestire le situazioni impreviste. In particolare usare l'eccezione `PortaMonetePienoException` affinché scatti nel metodo `aggiungi()` che avevamo definito nel modo seguente:

```
public void aggiungi(Moneta moneta) {
    System.out.println("Proviamo ad aggiungere una " +
      moneta.getDescrizione());
    int indiceLibero = primoIndiceLibero();
    if (indiceLibero == -1) {
        System.out.println("Portamonete pieno! La moneta " +
          moneta.getDescrizione() + " non è stata aggiunta...");
    } else {
        monete[indiceLibero] = moneta;
        System.out.println("E' stata aggiunta una " +
          moneta.getDescrizione());
    }
}
```

Fare in modo che l'eccezione `PortaMonetePienoException` sia lanciata opportunamente. Gestire anche eventuali altre eccezioni.

Esercizio 9.i)

Consideriamo i sorgenti creati con gli esercizi realizzati per il capitolo 6. Creare delle eccezioni personalizzate per gestire le situazioni impreviste. In particolare usare l'eccezione `MonetaNonTrovataException` affinché scatti nel metodo `preleva()` che avevamo definito nel modo seguente:

```
public Moneta preleva(Moneta moneta) {
    System.out.println("Proviamo a prelevare una " +
        moneta.getDescrizione());
    Moneta monetaTrovata = null;
    int indiceMonetaTrovata = indiceMonetaTrovata(moneta);
    if (indiceMonetaTrovata == -1) {
        System.out.println("Moneta non trovata!");
    } else {
        monetaTrovata = moneta;
        monete[indiceMonetaTrovata] = null;
        System.out.println("Una " + moneta.getDescrizione()
          + " prelevata");
    }
    return monetaTrovata;
}
```

Fare in modo che l'eccezione `MonetaNonTrovataException` sia lanciata opportunamente. Gestire anche eventuali altre eccezioni.

Esercizio 9.l)

Consideriamo i sorgenti creati con gli esercizi realizzati per il capitolo 6. Modificare la classe `TestMonete` per gestire correttamente l'eccezione `PortaMonetePienoException`.

Esercizio 9.m)

Aggiungere delle asserzioni nel costruttore della classe `PortaMonete`.

Esercizio 9.n)

Quali delle seguenti affermazioni sono corrette?

1. Le `RuntimeException` sono equivalenti alle unchecked exception.
2. `ArithmeticException` è una checked exception.
3. `ClassCastException` è una unchecked exception.
4. `NullPointerException` è una checked exception.

Esercizio 9.o)

Quali delle seguenti affermazioni sono corrette?

1. Nella clausola `throws` è possibile dichiarare solo le checked exception.
2. Nella clausola `throws` è possibile dichiarare solo le unchecked exception.
3. Nella clausola `throws` è possibile dichiarare una `NullPointerException`.
4. Con la clausola `throw` è possibile lanciare solo checked exception.
5. Con la clausola `throw` è possibile lanciare solo unchecked exception.
6. La clausola `throws` è obbligatoria se nel nostro metodo potrebbe essere lanciata una checked exception.
7. Un metodo che dichiara una clausola `throws` può essere invocato solo se si gestisce all'interno di un blocco `try catch`.

Esercizio 9.p)

Quali delle seguenti affermazioni sono corrette?

1. È possibile dichiarare solo checked exception.
2. Se dichiariamo una sottoclasse di `NullPointerException`, questa verrà lanciata insieme alla `NullPointerException`.
3. Se dichiariamo una sottoclasse di `NullPointerException`, questa verrà lanciata al posto della `NullPointerException`.
4. Se dichiariamo una sottoclasse di `ArithmeticException`, questa verrà lanciata nel caso ci sia un problema in un'operazione aritmetica.

Esercizio 9.q)

Non tenendo conto del costrutto try with resources, il blocco `finally` è obbligatorio (scegliere tutte le affermazioni valide):

1. Quando non ci sono blocchi `catch` dopo un blocco `try`.
2. Quando non ci sono blocchi `try` prima di un blocco `catch`.
3. Quando ci sono almeno due blocchi `catch` dopo un blocco `try`.
4. Mai.

Esercizio 9.r)

Considerando il seguente metodo:

```
public void metodoCheLanciaEccezione()
  throws ArrayIndexOutOfBoundsException {
    //INSERIRE CODICE QUI
}
```

Scegli tra i seguenti snippet quelli che potrebbero essere inseriti nel metodo `metodoCheLanciaEccezione()` affinché il codice precedente sia valido:

1. `throw new ArrayIndexOutOfBoundsException();`
2.
```
int i=0, j=0;
try {
  i = i/j;
} catch(ArithmeticException e) {
  throw new ArrayIndexOutOfBoundsException ();
}
```
3. `int i = 0;`
4. `System.out.println()`

Esercizio 9.s)

Considerando la seguente classe:

```
public class Esercizio9S {
    public static void main(String args[]) throws NullPointerException {
        Esercizio9S e = new Esercizio9S();
        e.metodo();
    }

    public NullPointerException metodo() throws Exception {
        String s = null;
        try {
            s.toString();
        } catch(ArithmeticException e) {
            throw new NullPointerException ();
        }
        return null;
    }
}
```

1. Scegliere tutte le affermazioni corrette:
2. Il codice non compila perché il metodo `metodo()` non può ritornare `NullPointerException`.
3. Il codice non compila perché il metodo `metodo()` ritorna `null` e non una `NullPointerException`.
4. Il codice non compila perché il metodo `main()` non dichiara la giusta eccezione nella sua clausola `throws`.
5. Il codice compila ma al runtime termina con una `NullPointerException`.
6. Il codice compila ma al runtime termina con una `Exception`.
7. Il codice compila ma al runtime termina con una `ArithmeticException`.

Esercizio 9.t)

Creare una classe `PortaAutomatica` che dichiari due metodi, `open()` e `close()`, dove quest'ultimo deve essere compatibile per essere chiamato con la tecnica del try with resources.

Esercizio 9.u)

Considerando la soluzione dell'esercizio 9.t (ovvero la classe `PortaAutomatica`): scrivere una semplice classe che ne testi il funzionamento mediante il costrutto try with resources.

Esercizio 9.v)

Riprendiamo l'esercizio 8.u dove avevamo verificato che le seguenti classi compilavano senza errori, ma che `SommaIntero` poteva lanciare un'eccezione al runtime.

```
public abstract class SommaNumero {
    public abstract Number somma (Number n1, Number n2);
}

public class SommaIntero extends SommaNumero {
    @Override
    public Integer somma(Number n1, Number n2) {
        return (Integer)n1 + (Integer)n2;
    }
}
```

Infatti con le seguenti istruzioni:

```
SommaIntero sommaIntero = new SommaIntero();
sommaIntero.somma(1.0, 1.0);
```

otterremo quest'eccezione al runtime:

```
Exception in thread "main" java.lang.ClassCastException:
  java.base/java.lang.Double cannot be cast to
  java.base/java.lang.Integer
    at SommaIntero.somma(SommaIntero.java:4)
    at SommaIntero.main(SommaIntero.java:9)
```

Nell'esercizio 8.v, avevamo chiesto di rendere robusta l'implementazione della classe `SommaIntero`, ed il risultato è stato il seguente:

```
public class SommaIntero extends SommaNumero {
    @Override
    public Integer somma(Number n1, Number n2) {
        if (n1 == null || n2 == null) {
            System.out.println("Impossibile sommare un operando null, "
              + "restituisco il valore di default");
            return Integer.MIN_VALUE;
        } else if (!(n1 instanceof Integer && n2 instanceof Integer)) {
            System.out.println("Passa solo variabili di tipo intere, "
              + "restituisco il valore di default");
            return Integer.MIN_VALUE;
        }
        return (Integer)n1 + (Integer)n2;
    }
}
```

Ora che si conosce la teoria delle eccezioni, riprogettare la classe `SommaIntero` usando la gestione delle eccezioni.

Esercizio 9.z)

Creare una semplice classe di test per la classe `SommaIntero` che abbiamo creato nell'esercizio 9.v.

Soluzioni degli esercizi del capitolo 9

Soluzione 9.a) Gestione delle eccezioni e degli errori, Vero o Falso:

1. **Vero**, perché `ArithmeticException` è sottoclasse di `RuntimeException`.
2. **Falso**.
3. **Falso**, produrrà il seguente output:

```
Divisione per zero...
Finally!
```

4. **Falso**, produrrà un errore in compilazione (l'ordine dei blocchi `catch` non è regolare).
5. **Falso**.
6. **Falso**, solo le sottoclassi di `Throwable`.
7. **Vero**.
8. **Falso**.
9. **Vero**.
10. **Vero**.

Soluzione 9.b) Gestione delle asserzioni, Vero o Falso:

1. **Vero**.
2. **Vero**.

3. **Falso**.
4. **Vero**.
5. **Vero**.
6. **Vero**.
7. **Vero**.
8. **Vero**.
9. **Vero**.
10. **Vero**.

Soluzione 9.c)

Come abbiamo visto nel paragrafo 9.6.3.1, non dovremmo mai usare le asserzioni per testare i parametri di un metodo pubblico. Quindi indubbiamente è più corretto usare la gestione delle eccezioni.

Soluzione 9.d)

Una possibile implementazione potrebbe essere la seguente:

```
public void aggiungiTesto(String testo) throws RuntimeException {
    if (contenuto == null) {
        contenuto = "";
    }
    if (testo == null) {
        throw new RuntimeException("testo = null");
    }
    contenuto += testo;
}
```

Si noti che abbiamo rilanciato una `RuntimeException`, ma avremmo potuto rilanciare una qualsiasi altra eccezione (per esempio `Exception` stessa). Inoltre la clausola `throws` accanto alla dichiarazione del metodo non è tecnicamente obbligatoria ma consigliabile.

Soluzione 9.e)

Il listato potrebbe essere simile al seguente:

```
public void aggiungiTesto(String testo, int posizione) {
    try {
        if (testo != null) {
            contenuto = contenuto.substring(0, posizione) + testo
```

```
                    + contenuto.substring(posizione);
                }
            } catch (NullPointerException exc) {
                System.out.println("Il contenuto è null : "
                  + exc.getMessage());
                contenuto = "" + testo;
            } catch (StringIndexOutOfBoundsException exc) {
                System.out.println("L'indice " + posizione
                  + " è invalido : " + exc.getMessage());
                contenuto = (posizione < 0 ? testo + contenuto : contenuto
                  + testo);
            }
        }
```

In questo esempio abbiamo solo controllato se il testo da aggiungere è `null`, in tal caso nessuna operazione viene eseguita. Poi abbiamo gestito la `NullPointerException` che si presenterebbe nel caso `contenuto` valga `null`. Nella clausola `catch` abbiamo stampato un messaggio significativo e mantenuto la coerenza con il metodo precedentemente presentato.
Abbiamo anche gestito una `StringIndexOutOfBoundsException` che scatterebbe se la posizione specificata contenesse un numero negativo o maggiore della dimensione del contenuto del file. Anche in questo caso nella clausola `catch` abbiamo dapprima stampato un messaggio significativo, e poi "accomodato" la situazione. In particolare abbiamo fatto in modo che (sfruttando anche un operatore ternario), se la variabile `posizione` è specificata con un valore negativo, allora la variabile `testo` viene messa all'inizio prima della variabile `contenuto` (come se fosse stata specificata la posizione 0). Se invece viene impostata con un valore superiore all'ultimo indice disponibile per il contenuto, allora la variabile `testo` viene aggiunta alla fine del contenuto.
Infine, per quanto riguarda il `finally`, l'impostazione della nostra soluzione non ne richiede l'uso.

Soluzione 9.f)

Il listato della classe `TestFileSorgente` potrebbe essere il seguente:

```
public class TestFileSorgente {
    public static void main(String args[]) {
        FileSorgente fileSorgente = new FileSorgente("Test.java",
          TipoFile.JAVA, "public class MyClass {\n\r");
        System.out.println(fileSorgente.getContenuto());
        // Test aggiungiTesto (String) corretto
        fileSorgente.aggiungiTesto("}");
        System.out.println(fileSorgente.getContenuto());
        // Test aggiungiTesto (String,int) corretto
```

```
        fileSorgente.aggiungiTesto("//Test aggiunta testo\n\r", 23);
        System.out.println(fileSorgente.getContenuto());
        // Test aggiungiTesto (String,int) scorretto
        fileSorgente.aggiungiTesto("//Test aggiunta testo\n\r", -1);
        System.out.println(fileSorgente.getContenuto());
        // Test aggiungiTesto (String,int) scorretto
        fileSorgente.aggiungiTesto("//Test aggiunta testo\n\r", 100);
        System.out.println(fileSorgente.getContenuto());
        FileSorgente fileSorgenteVuoto =
          new FileSorgente("FileVuoto.c", TipoFile.C);
        fileSorgenteVuoto.aggiungiTesto("//Test aggiunta testo\n\r", 3);
        System.out.println(fileSorgenteVuoto.getContenuto());
        FileSorgente fileSorgenteVuoto2 =
          new FileSorgente("FileVuoto2.cpp", TipoFile.C_PLUS_PLUS);
        fileSorgenteVuoto2.aggiungiTesto("//Test aggiunta testo\n\r");
    }
}
```

Soluzione 9.g)

Il nuovo listato della classe `PortaMonetePienoException` potrebbe essere il seguente:

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

Il listato del costruttore della classe `PortaMonete`, con i nuovi requisiti potrebbe essere trasformato nel seguente modo:

```
    public PortaMonete(int... valori) {
        try {
            int numeroMonete = valori.length;
            for (int i = 0; i < numeroMonete; i++) {
                if (i >= 10) {
                    throw new PortaMonetePienoException (
                      "Sono state inserite solo le prime 10 monete!");
                }
                monete[i] = new Moneta(valori[i]);
            }
        } catch (PortaMonetePienoException | NullPointerException exc) {
            System.out.println(exc.getMessage());
        }
    }
```

Si noti che passando `null` come parametro al costruttore, potremmo generare una `NullPointerException` quando viene richiesta la variabile `length` sul parame-

tro `valori` (che varrebbe appunto `null`).
Con un "multicatch" abbiamo garantito il funzionamento del costruttore senza che il programma si interrompa, stampando il messaggio del problema che si è presentato. Tuttavia non è corretto in questo caso gestire allo stesso modo questi due tipi di eccezione, in quanto nel caso della `NullPointerException` il messaggio stampato sarà semplicemente:

```
null
```

che è poco esplicativo!
Sarebbe meglio gestire le due eccezioni in questo modo:

```
public PortaMonete(int... valori) {
    try {
        int numeroMonete = valori.length;
        for (int i = 0; i < numeroMonete; i++) {
            if (i >= 10) {
                throw new PortaMonetePienoException (
                  "Sono state inserite solo le prime 10 monete!");
            }
            monete[i] = new Moneta(valori[i]);
        }
//  } catch (PortaMonetePienoException | NullPointerException exc) {
    } catch (PortaMonetePienoException exc) {
        System.out.println(exc.getMessage());
    } catch (NullPointerException exc) {
        System.out.println("Il portamonete è stato creato vuoto");
    }
}
```

Soluzione 9.h)

Il listato del metodo `aggiungi()` della classe `PortaMonete` con i nuovi requisiti, potrebbe essere trasformato nel seguente modo:

```
public void aggiungi(Moneta moneta) throws PortaMonetePienoException {
    try {
        System.out.println("Proviamo ad aggiungere una " +
          moneta.getDescrizione());
    } catch (NullPointerException exc) {
        throw new NullPointerException("La moneta passata era null");
    }
    int indiceLibero = primoIndiceLibero();
    if (indiceLibero == -1) {
        throw new PortaMonetePienoException(
          "Portamonete pieno! La moneta "
          + moneta.getDescrizione() + " non è stata aggiunta...");
```

```
        } else {
            monete[indiceLibero] = moneta;
            System.out.println("E' stata aggiunta una " +
              moneta.getDescrizione());
        }
    }
```

Si noti che abbiamo gestito anche la `NullPointerException` catturandola e rilanciandola con un messaggio migliore del "`null`". Anche in questo caso l'eccezione si verificherebbe nel caso il parametro passato fosse `null`, non appena fosse chiamato il metodo `getDescrizione()` sull'oggetto `moneta` (che varrebbe `null`).

Soluzione 9.i)

L'eccezione richiesta potrebbe essere la seguente:

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

Il listato del metodo `preleva()` della classe `PortaMonete`, con i nuovi requisiti, potrebbe essere trasformato nel seguente modo:

```
    public Moneta preleva(Moneta moneta)
      throws MonetaNonTrovataException {
        try {
        System.out.println("Proviamo a prelevare una " +
          moneta.getDescrizione());
        } catch (NullPointerException exc) {
            throw new MonetaNullException();
        }
        Moneta monetaTrovata = null;
        int indiceMonetaTrovata = indiceMonetaTrovata(moneta);
        if (indiceMonetaTrovata == -1) {
            throw new MonetaNonTrovataException("Moneta non trovata!");
        } else {
            monetaTrovata = moneta;
            monete[indiceMonetaTrovata] = null;
            System.out.println("Una " + moneta.getDescrizione()
              + " prelevata");
        }
        return monetaTrovata;
    }
```

Si noti che abbiamo gestito anche la `NullPointerException` catturandola e rilanciandola con un messaggio migliore del "`null`". Anche in questo caso l'eccezione si verificherebbe nel caso il parametro passato fosse `null`, non appena viene chiama-

to il metodo `getDescrizione()` sull'oggetto `moneta` (che varrebbe `null`). Possiamo migliorare questo meccanismo creando un'altra eccezione personalizzata:

```
public class MonetaNullException extends RuntimeException {
    public MonetaNullException() {
        super("La moneta passata era null");
    }
}
```

A `MonetaNullException` gli abbiamo fatto estendere `RuntimeException`, e quindi si tratta di una unchecked exception che non ha bisogno di essere dichiarata nella clausola throws del metodo che la lancia. E quindi il metodo `aggiungi()` può essere modificato come segue:

```
    public void aggiungi(Moneta moneta)
      throws PortaMonetePienoException {
        try {
            System.out.println("Proviamo ad aggiungere una " +
              moneta.getDescrizione());
        } catch (NullPointerException exc) {
            throw new MonetaNullException();
        }
        int indiceLibero = primoIndiceLibero();
        if (indiceLibero == -1) {
            throw new PortaMonetePienoException(
              "Portamonete pieno! La moneta "
              + moneta.getDescrizione() + " non è stata aggiunta...");
        } else {
            monete[indiceLibero] = moneta;
            System.out.println("E' stata aggiunta una " +
              moneta.getDescrizione());
        }
    }
```

Soluzione 9.l)

Equivalentemente all'esercizio precedente il listato richiesto potrebbe essere il seguente:

```
/**
 * Classe di test per la classe Moneta.
 *
 * @author Claudio De Sio Cesari
 */
public class TestMonete {

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

```
        Moneta monetaDaVentiCentesimi = new Moneta(20);
        Moneta monetaDaUnCentesimo = new Moneta(1);
        Moneta monetaDaUnEuro = new Moneta(100);
        // Creaiamo un portamonete con 11 monete
        PortaMonete portaMoneteInsufficiente =
          new PortaMonete(2, 5, 100, 10, 50, 10, 100, 200, 10, 5, 2);
        // Creaiamo un portamonete con 8 monete
        PortaMonete portaMonete =
          new PortaMonete(2, 5, 100, 10, 50, 10, 100, 200);
        portaMonete.stato();

        try {
            // Aggiungiamo una moneta da 20 centesimi
            portaMonete.aggiungi(monetaDaVentiCentesimi);
        } catch (PortaMonetePienoException | MonetaNullException exc) {
            System.out.println(exc.getMessage());
        }

        try {
            // Aggiungiamo la decima moneta da 1 centesimo.
            portaMonete.aggiungi(monetaDaUnCentesimo);
        } catch (PortaMonetePienoException | MonetaNullException exc) {
            System.out.println(exc.getMessage());
        }

        try {
            // Aggiungiamo l'undicesima moneta (dovremmo ottenere un
            // errore e la moneta non sarà aggiunta)
            portaMonete.aggiungi(monetaDaUnEuro);
        } catch (PortaMonetePienoException | MonetaNullException exc) {
            System.out.println(exc.getMessage());
        }

        // Valutiamo lo stato del portamonete.
        portaMonete.stato();

        try {
            // preleviamo 20 centesimi
            portaMonete.preleva(monetaDaVentiCentesimi);
        } catch (MonetaNonTrovataException exc) {
            System.out.println(exc.getMessage());
        }

        try {
            // Aggiungiamo l'undicesima moneta (dovremmo ottenere un
            // errore e la moneta non sarà aggiunta)
            portaMonete.aggiungi(monetaDaUnEuro);
        } catch (PortaMonetePienoException | MonetaNullException exc) {
            System.out.println(exc.getMessage());
        }
```

```
            portaMonete.stato();

            try {
                //Cerchiamo una moneta non esistente (dovremmo ottenere
                //una stampa di errore)
                portaMonete.preleva(new Moneta(7));
            } catch (MonetaNonTrovataException exc) {
                System.out.println(exc.getMessage());
            }

            //testiamo il passaggio di null al costruttore del porta monete
            PortaMonete portaMoneteEccezione = new PortaMonete(null);
            portaMonete.stato();

            try {
                // proviamo ad aggiungere null
                portaMonete.aggiungi(null);
            } catch (PortaMonetePienoException | MonetaNullException exc) {
                System.out.println(exc.getMessage());
            }

            try {
                //Proviamo a prelevare una moneta null
                portaMonete.preleva(null);
            } catch (MonetaNonTrovataException | MonetaNullException exc) {
                System.out.println(exc.getMessage());
            }
        }
    }
```

Il cui output sarà:

```
Creata una moneta da 20 centesimi di EURO
Creata una moneta da 1 centesimo di EURO
Creata una moneta da 1 EURO
Creata una moneta da 2 centesimi di EURO
Creata una moneta da 5 centesimi di EURO
Creata una moneta da 1 EURO
Creata una moneta da 10 centesimi di EURO
Creata una moneta da 50 centesimi di EURO
Creata una moneta da 10 centesimi di EURO
Creata una moneta da 1 EURO
Creata una moneta da 2 EURO
Creata una moneta da 10 centesimi di EURO
Creata una moneta da 5 centesimi di EURO
Sono state inserite solo le prime 10 monete!
Creata una moneta da 2 centesimi di EURO
Creata una moneta da 5 centesimi di EURO
Creata una moneta da 1 EURO
Creata una moneta da 10 centesimi di EURO
Creata una moneta da 50 centesimi di EURO
```

```
Creata una moneta da 10 centesimi di EURO
Creata una moneta da 1 EURO
Creata una moneta da 2 EURO
Il portamonete contiene:
Una moneta da 2 centesimi di EURO
Una moneta da 5 centesimi di EURO
Una moneta da 1 EURO
Una moneta da 10 centesimi di EURO
Una moneta da 50 centesimi di EURO
Una moneta da 10 centesimi di EURO
Una moneta da 1 EURO
Una moneta da 2 EURO
Proviamo ad aggiungere una moneta da 20 centesimi di EURO
E' stata aggiunta una moneta da 20 centesimi di EURO
Proviamo ad aggiungere una moneta da 1 centesimo di EURO
E' stata aggiunta una moneta da 1 centesimo di EURO
Proviamo ad aggiungere una moneta da 1 EURO
Portamonete pieno! La moneta moneta da 1 EURO non è stata aggiunta...
Il portamonete contiene:
Una moneta da 2 centesimi di EURO
Una moneta da 5 centesimi di EURO
Una moneta da 1 EURO
Una moneta da 10 centesimi di EURO
Una moneta da 50 centesimi di EURO
Una moneta da 10 centesimi di EURO
Una moneta da 1 EURO
Una moneta da 2 EURO
Una moneta da 20 centesimi di EURO
Una moneta da 1 centesimo di EURO
Proviamo a prelevare una moneta da 20 centesimi di EURO
Una moneta da 20 centesimi di EURO prelevata
Proviamo ad aggiungere una moneta da 1 EURO
E' stata aggiunta una moneta da 1 EURO
Il portamonete contiene:
Una moneta da 2 centesimi di EURO
Una moneta da 5 centesimi di EURO
Una moneta da 1 EURO
Una moneta da 10 centesimi di EURO
Una moneta da 50 centesimi di EURO
Una moneta da 10 centesimi di EURO
Una moneta da 1 EURO
Una moneta da 2 EURO
Una moneta da 1 EURO
Una moneta da 1 centesimo di EURO
Creata una moneta da 7 centesimi di EURO
Proviamo a prelevare una moneta da 7 centesimi di EURO
Moneta non trovata!
Il portamonete è stato creato vuoto
Il portamonete contiene:
Una moneta da 2 centesimi di EURO
Una moneta da 5 centesimi di EURO
```

```
Una moneta da 1 EURO
Una moneta da 10 centesimi di EURO
Una moneta da 50 centesimi di EURO
Una moneta da 10 centesimi di EURO
Una moneta da 1 EURO
Una moneta da 2 EURO
Una moneta da 1 EURO
Una moneta da 1 centesimo di EURO
La moneta passata era null
La moneta passata era null
```

Soluzione 9.m)

Potremmo modificare il listato di `PortaMonete` nel seguente modo:

```
public class PortaMonete {

    private static final int DIMENSIONE = 10;

    private final Moneta[] monete = new Moneta[DIMENSIONE];


    public PortaMonete(int... valori) {
        assert monete.length == DIMENSIONE;
        try {
            int numeroMonete = valori.length;
            for (int i = 0; i < numeroMonete; i++) {
                if (i >= DIMENSIONE) {
                    throw new PortaMonetePienoException(
                      "Sono state inserite solo le prime + "
                      + DIMENSIONE + " monete!");
                }
                monete[i] = new Moneta(valori[i]);
            }
   //   } catch (PortaMonetePienoException | NullPointerException exc) {
        } catch (PortaMonetePienoException exc) {
            System.out.println(exc.getMessage());
        } catch (NullPointerException exc) {
            System.out.println("Il portamonete è stato creato vuoto");
        }
        assert monete.length == DIMENSIONE;
    }
//...
```

Si noti che abbiamo definito la costante `DIMENSIONE` impostata a `10`. Poi come asserzioni abbiamo usato una pre-condizione e una post-condizione. In entrambi i casi abbiamo asserito lo stesso concetto: la lunghezza dell'array di monete è uguale al valore di `DIMENSIONE`. Si noti anche come con queste semplici asserzioni stiamo

rinforzando la logica del costruttore, è come se stessimo dicendo "qualsiasi cosa succeda, il valore della lunghezza dell'array non può cambiare". In questo modo il nostro codice rimarrà coerente con questa asserzione attraverso tutte le modifiche che verranno fatte in seguito.

Soluzione 9.n)

Solo l'affermazione numero 3 è corretta.

Soluzione 9.o)

Sono corrette le affermazioni numero 3 e numero 6. Infatti la clausola `throws` e il comando `throw` possono essere utilizzati per ogni tipo di eccezione, e questo esclude che siano corrette le affermazioni 1, 2, 4 e 5. L'affermazione 7 è invece scorretta perché il metodo che invoca un metodo che dichiara la clausola `throws`, potrebbe a sua volta dichiarare la clausola `throws`!

Soluzione 9.p)

Nessuna delle affermazioni è corretta.

Soluzione 9.q)

Solo la prima affermazione è corretta.

Soluzione 9.r)

Tutti gli snippet sono validi perché, nella clausola throws, viene dichiarata la `ArrayIndexOutOfBoundsException` che è un'eccezione unchecked, e che quindi non è obbligatoria esplicitamente. Si noti che la risposta 2 fa scattare una `ArithmeticException`, la gestisce e rilancia una `ArrayIndexOutOfBoundsException`.

Soluzione 9.s)

L'unica affermazione corretta è la 3. La sua correttezza esclude la correttezza delle affermazioni 4, 5 e 6. La 1 e la 2 sono false perché `NullPointerException` è comunque una classe.

Soluzione 9.t)

Una semplice implementazione potrebbe essere la seguente:

```
public class PortaAutomatica implements AutoCloseable {
    public void close() {
        System.out.println("La porta si sta chiudendo");
    }

    public void open() {
        System.out.println("La porta si sta aprendo");
    }
}
```

Soluzione 9.u)

La soluzione potrebbe essere semplice come la seguente:

```
public class Esercizio9T {
    public static void main(String args[]) {
        try (PortaAutomatica portaAutomatica = new PortaAutomatica();) {
            portaAutomatica.open();
        }
    }
}
```

Il cui output, una volta eseguita, sarà:

```
La porta si sta aprendo
La porta si sta chiudendo
```

Soluzione 9.v)

La soluzione con la gestione delle eccezioni è indubbiamente più semplice ed elegante:

```
public class SommaIntero extends SommaNumero {
    @Override
    public Integer somma(Number n1, Number n2) {
        Integer risultato = null;
        try {
            risultato = (Integer)n1 + (Integer)n2;
        } catch (NullPointerException e) {
            System.out.println("Impossibile sommare un operando null");
        } catch (ClassCastException e) {
            System.out.println("Passa solo variabili di tipo intere");
        }
        return risultato;
    }
}
```

Soluzione 9.z)

Con la seguente classe di test:

```
public class Esercizio9Z {
    private static final String PREFISSO_FRASE ="Il risultato è ";
    public static void main(String args[]) {
        SommaIntero sommaIntero = new SommaIntero();
        System.out.println(PREFISSO_FRASE + sommaIntero.somma(1.0, 1.0));
        System.out.println(PREFISSO_FRASE + sommaIntero.somma(1, null));
        System.out.println(PREFISSO_FRASE + sommaIntero.somma(1, 25));
    }
}
```

Possiamo verificare i casi possibili che abbiamo previsto.

Esercizi del capitolo 10

Enumerazioni e Tipi Innestati

Il capitolo 10 è il primo dei capitoli della terza parte del libro "Caratteristiche avanzate". In particolare i tipi innestati sono a tutti gli effetti uno degli argomenti più complessi di Java, per la loro sintassi, e soprattutto per le astruse proprietà che li caratterizzano. Padroneggiare i tipi innestati permetterà di interfacciarci con il linguaggio in maniera più efficace. Per quanto riguarda le enumerazioni, tutto sommato si tratta di un argomento non così complesso. Ci sono alcune regole particolari che bisogna comprendere, ma l'importante è capire quando conviene utilizzarle. Anche per questo capitolo troverete tanti esercizi a risposta multipla (alcuni molto difficili) che supportano alla preparazione dell'esame di certificazione Oracle, ma anche diverse implementazioni da codificare.

Esercizio 10.a) Tipi innestati, Vero o Falso:

1. Una classe innestata è una classe che viene dichiarata all'interno di un'altra classe.
2. Una classe anonima è anche innestata, ma non ha nome. Inoltre, per essere dichiarata, deve per forza essere istanziata.
3. Le classi innestate non sono necessarie per l'Object Orientation.
4. Una classe innestata deve essere per forza istanziata.
5. Per istanziare una classe innestata pubblica a volte bisogna istanziare prima la classe esterna.

6. Una classe innestata dichiarata `private` deve dichiarare anche i metodi "set" e "get" per poter essere utilizzata da una terza classe.
7. Una classe innestata non può avere lo stesso nome della classe che la contiene.
8. Una classe anonima può avere lo stesso nome della classe che la contiene.
9. Una classe innestata può accedere a membri statici della classe che la contiene solo se è dichiarata statica.
10. Una classe innestata non può essere dichiarata astratta.

Esercizio 10.b) Enumerazioni, Vero o Falso:

1. Le enumerazioni non si possono istanziare se non all'interno della definizione dell'enumerazione stessa. Infatti, possono avere solamente costruttori `private`.
2. Le enumerazioni possono dichiarare metodi e possono essere estese da classi che possono sottoporne a override i metodi. Non è però possibile che una `enum` estenda un'altra `enum`.
3. Il metodo `values()` appartiene ad ogni enumerazione ma non alla classe `java.lang.Enum`.
4. Il seguente codice viene compilato senza errori:

```
public enum MyEnum {
  public void metodo1() {

  }
  public void metodo2() {

  }
  ENUM1, ENUM2;
}
```

5. Il seguente codice viene compilato senza errori:

```
public enum MyEnum {
  ENUM1 {
    public void metodo() {

    }
  }, ENUM2;

  public void metodo2() {
```

```
  }
}
```

6. Il seguente codice viene compilato senza errori:

```
public enum MyEnum {
  ENUM1 (), ENUM2;
  private MyEnum(int i) {

  }
}
```

7. Il seguente codice viene compilato senza errori:

```
public class Volume {
  public enum Livello {
    ALTO, MEDIO, BASSO
  } ;
  // implementazione della classe . . .
  public static void main(String args[]) {
    switch (getLivello()) {
      case ALTO:
      System.out.println(Livello.ALTO);
      break;
      case MEDIO:
      System.out.println(Livello.MEDIO);
      break;
      case BASSO:
      System.out.println(Livello.BASSO);
      break;
    }
  }
  public static Livello getLivello() {
    return Livello.ALTO;
  }
}
```

8. Se dichiariamo la seguente enumerazione:

```
public enum MyEnum {
  ENUM1 {
    public void metodo1() {

    }
  },
  ENUM2 {
    public void metodo2() {

    }
  }
}
```

il seguente codice potrebbe essere correttamente compilato:

```
MyEnum.ENUM1.metodo1();
```

9. Non è possibile dichiarare enumerazioni con un unico elemento.

10. Si possono innestare enumerazioni in enumerazioni in questo modo:

```
public enum MyEnum {
  ENUM1 (), ENUM2;
  public enum MyEnum2 {a,b,c}
}
```

ed il seguente codice viene compilato senza errori:

```
System.out.println(MyEnum.MyEnum2.a);
```

Esercizio 10.c)

Modificare i sorgenti creati con gli esercizi realizzati per il capitolo 8, dopo aver sostituito l'interfaccia `TipoFile`, creata nell'esercizio 8.c, con un'enumerazione. Tutto dovrà funzionare come in precedenza.

Esercizio 10.d)

Partendo dalla soluzione dell'esercizio precedente, innestare l'enumerazione `TipoFile` nella classe `File`. Che modifiche bisogna apportare all'applicazione per continuare a farla funzionare come prima?

Esercizio 10.e)

Consideriamo i sorgenti creati con gli esercizi realizzati per il capitolo 6, e modificati con gli esercizi del capitolo 9. La classe `Moneta` potrebbe essere attualmente istanziata in maniera scorretta e bisognerebbe gestire eventuali eccezioni per esempio se come parametro al costruttore della classe `Moneta` fosse passato un valore negativo. Segue il codice che abbiamo sino ad ora sviluppato (commenti omessi):

```
public class Moneta {

    public final static String VALUTA = "EURO";

    private final int valore;

    public Moneta(int valore) {
        this.valore = valore;
        System.out.println("Creata una " + getDescrizione());
```

```
    }

    public int getValore() {
        return valore;
    }

    public String getDescrizione() {
        String descrizione = "moneta da "
          + formattaStringaDescrittiva(valore) + VALUTA;
        return descrizione;
    }

    private String formattaStringaDescrittiva(int valore) {
        String stringaFormattata = " centesimi di ";
        if (valore == 1) {
            stringaFormattata = " centesimo di ";
        } else if (valore > 99) {
            stringaFormattata = " ";
            valore /= 100;
        }
        return valore + stringaFormattata;
    }
}
```

Creare un'enumerazione che consenta di rendere il costruttore (e quindi l'intero programma) più robusto, e che ci permetta di evitare la gestione delle eccezioni. Riscrivere anche la classe `Moneta`.

Esercizio 10.f)

In base all'esercizio precedente, modificare la classe `PortaMonete` di conseguenza.

Esercizio 10.g)

In base ai due ultimi esercizi, modificare la classe `TestMonete` di conseguenza.

Esercizio 10.h)

Supponiamo di voler scrivere un programma e di voler risfruttare la seguente classe `Persona`, ereditata da un programma già scritto e non modificabile.

```
public class Persona {

    private String nome;
    private String cognome;
    private String dataDiNascita;
    private String professione;
    private String indirizzo;
```

```
public Persona(String nome, String cognome) {
    this.nome = nome;
    this.cognome = cognome;
}

public Persona(String nome, String cognome, String dataDiNascita,
  String professione, String indirizzo) {
    this.nome = nome;
    this.cognome = cognome;
    this.dataDiNascita = dataDiNascita;
    this.professione = professione;
    this.indirizzo = indirizzo;
}

public String getNome() {
    return nome;
}

public void setNome(String nome) {
    this.nome = nome;
}

public String getCognome() {
    return cognome;
}

public void setCognome(String cognome) {
    this.cognome = cognome;
}

public String getDataDiNascita() {
    return dataDiNascita;
}

public void setDataDiNascita(String dataDiNascita) {
    this.dataDiNascita = dataDiNascita;
}

public String getProfessione() {
    return professione;
}

public void setProfessione(String professione) {
    this.professione = professione;
}

public String getIndirizzo() {
    return indirizzo;
}
```

```
    public void setIndirizzo(String indirizzo) {
        this.indirizzo = indirizzo;
    }

    @Override
    public String toString() {
        return "Persona{" + "nome=" + nome + ", cognome="
          + cognome + '}';
    }
}
```

Purtroppo nel nostro contesto, avremmo bisogno di ridefinire il metodo `toString()` in modo che stampi non solo le informazioni sul nome e il cognome della persona, ma anche la data di nascita, l'indirizzo e la professione. Come già detto però, la classe è già in uso e non è possibile modificarla. In particolare il nostro requisito è che il metodo `toString()` restituisca la seguente stringa:

```
Nome:                   Arjen Anthony
Cognome:                Lucassen
Professione:            Compositore
Data di Nascita:   03/04/1960
Indirizzo:         Olanda
```

Si crei quindi una classe `TestPersona` che ridefinisca in qualche modo il metodo `toString()` della classe `Persona` e stampi l'output di cui sopra.

Per la formattazione è possibile sfruttare il carattere di escape `\t` introdotto nel paragrafo 3.3.5.

Esercizio 10.i)

Data la seguente classe:

```
public class Esterna {
    private int intero = 5;

    public static void main(String[] args) {
        Esterna.Interna interna = new Esterna().new Interna();
        interna.metodoInterno();
    }

    private class Interna {
        private int intero = 10;
        protected void metodoInterno() {
            System.out.println(super.intero);
```

```
            }
        }
    }
```

Quali delle seguenti affermazioni sono vere?

1. Eseguita la classe `Esterna`, viene lanciata una `NullPointerException`.
2. Eseguita la classe `Esterna`, viene stampato `50`.
3. Eseguita la classe `Esterna`, viene stampato `10`.
4. Eseguita la classe `Interna`, viene stampato `5`.
5. Eseguita la classe `Interna`, viene stampato `0`.
6. Impossibile compilare questo codice.

Esercizio 10.l)

Tenendo conto della risposta dell'esercizio precedente, modificare il codice (con una modifica minimale) in modo tale da far stampare il valore `2`. Poi modificarlo nuovamente per fargli stampare anche il valore `4`.

Esercizio 10.m)

Data la seguente classe:

```
public class Esterna {
    public int a = 1;
    private int b = 2;

    public void metodoEsterno(final int c) {
        int d = 4;
        class Interna {
            private void metodoInterno(int e) {

            }
        }
    }
}
```

Quali delle seguenti affermazioni sono corrette?

1. All'interno del metodo `metodoInterno` è possibile referenziare la variabile `a`.
2. All'interno del metodo `metodoInterno` è possibile referenziare la variabile `b`.
3. All'interno del metodo `metodoInterno` è possibile referenziare la variabile `c`.

4. All'interno del metodo `metodoInterno` è possibile referenziare la variabile `d`.
5. All'interno del metodo `metodoInterno` è possibile referenziare la variabile `e`.
6. La classe non può essere compilata.

Esercizio 10.n)

Data la seguente classe:

```
public class Esercizio10N {

    public static void main(String args[]) {
        new Interfaccia() {
            public void metodo() {
                System.out.println("Classe anonima");
            }
        }.metodo();
    }

    private interface Interfaccia {
        void metodo();
    }
}
```

Quali delle seguenti affermazioni è vera?

1. Eseguita la classe `Esercizio10N`, viene lanciata una `NullPointerException`.
2. Eseguita la classe `Esercizio10N`, viene lanciata una `CannotInstantiateInterfaceException`.
3. Eseguita la classe `Esercizio10N`, viene stampato "Classe Anonima".
4. Eseguita la classe `Esercizio10N`, viene stampato "Null".
5. Impossibile compilare questo codice.

Esercizio 10.o)

Data la seguente classe:

```
public class Esterna {
    private class Interna {
        private static int effectivelyFinalVariable = 10;

        Interna() {
            effectivelyFinalVariable = 11;
```

```
        }

        protected void metodo () {
            System.out.println(effectivelyFinalVariable);
        }
    }
}
```

Quali delle seguenti affermazioni sono vere?

1. Impossibile compilare questo codice.
2. La variabile `effectivelyFinalVariable` non è "effettivamente `final`".
3. La variabile `effectivelyFinalVariable` non ha importanza che sia o meno "effettivamente `final`" perché è una variabile d'istanza.
4. La variabile `effectivelyFinalVariable` non ha importanza che sia o meno "effettivamente `final`" perché appartiene alla classe interna e non alla classe esterna.

Esercizio 10.p)

Data la seguente classe:

```
public class Esterna {
    private final static String stringa = "Classe Innestata";

    protected Esterna () {
        private static class Innestata {
            protected void metodo () {
                System.out.println(stringa);
            }
        }
    }
}
```

Quali delle seguenti affermazioni sono vere?

1. Impossibile compilare questo codice.
2. La costante statica `stringa` non è "effettivamente `final`" perché è anche statica, e quindi non può essere utilizzata all'interno della classe interna `Innestata`.
3. La costante statica `stringa` non è accessibile al metodo `metodo()` perché dichiarata statica.

4. La classe `Interna`, essendo dichiarata all'interno di un costruttore `protected`, può essere utilizzata solo all'interno del costruttore.

Esercizio 10.q)

Dato il seguente codice:

```
public enum Esercizio10Q {
    A, B, C;

    private enum InnerEnum {
        C, D, E;

        protected enum InnerInnerEnum {
            F,G,H
        }
    }


    public static void main(String args[]) {
        System.out.println(/*INSERISCI CODICE QUI*/);
    }
}
```

Quali delle seguenti espressioni è possibile inserire al posto del commento `/*INSERISCI CODICE QUI*/` per stampare il valore `H` (è possibile scegliere zero o più espressioni):

1. `Esercizio10Q.A.InnerEnum.C.InnerInnerEnum.H`
2. `Esercizio10Q.InnerEnum.InnerInnerEnum.H`
3. `InnerInnerEnum.H`
4. `InnerEnum.A.InnerInnerEnum.H`
5. `InnerEnum.InnerInnerEnum.F.G.H`
6. `InnerEnum.InnerInnerEnum.H`
7. `Esercizio10Q.A.C.H`
8. Nessuno: il codice non compila.

Esercizio 10.r)

Dato il seguente codice:

```
public enum Esercizio10R implements Interface {
    UNO {
```

```
        @Override
        public int metodo() {
            return 29 + 7 + 74;
        }
    },
    DUE,
    TRE{
        @Override
        public int metodo() {
            return 12 + 11 + 6;
        }
    };
    @Override
    public int metodo() {
        return 14 + 4 + 4;
    }

    public static void main(String args[]) {
        Interface i = Esercizio10R.TRE;
        System.out.println(i.metodo());
    }
}

interface Interface {
    int metodo();
}
```

Se viene eseguito il file `Esercizio10R`, cosa verrà stampato (scegliere una sola risposta)?

1. `22`
2. `29`
3. `110`
4. `161`
5. `TRE`
6. `TRE.metodo()`
7. `i.metodo()`
8. Niente, il codice non compila.

Esercizio 10.s)

Dato questo codice:

```
public enum Esercizio10S {
```

```
    A, B, C;
    public class Interna {
        public enum EnumInterna {
            D, E, F;
        }
    }
    public static void main(String args[]) {
        for (Esercizio10S.Interna.EnumInterna item :
                    Esercizio10S.Interna.EnumInterna.values()) {
            System.out.println(item);
        }
    }
}
```

Una volta eseguita l'enumerazione `Esercizio10S`, l'output sarà:

1. A, B, C
2. D, E, F
3. A B C
4. D E F
5. ABC
6. DEF
7. Nessuna delle precedenti: viene lanciata una `NullPointerException`.
8. Nessuna delle precedenti: il codice non compila per un errore nel metodo `main()`.
9. Nessuna delle precedenti: il codice non compila per un errore nella classe `Interna`.

Esercizio 10.t)

Dopo aver letto la soluzione dell'esercizio precedente (in particolare l'output della compilazione), si modifichi il codice per renderlo compilabile, risolvendo i vari errori che si presentano dopo le varie compilazioni. Eseguendo la versione compilabile definitiva il programma dovrà stampare il seguente output:

```
D
E
F
```

Questo esercizio testa la vostra capacità di interpretare i messaggi di errore del compilatore, e risolverli. Praticamente quello che ogni programmatore fa in continuazione!

Esercizio 10.u)

Cosa dobbiamo utilizzare per far eseguire algoritmi diversi (anche parziali) ad un metodo (scegliere tutte le risposte corrette) senza aver prima creato il codice dell'algoritmo?

1. Un'oggetto istanziato da una classe già esistente.
2. Un'interfaccia.
3. Un'enumerazione.
4. Una classe interna.
5. Una classe innestata.
6. Una classe anonima.
7. Un costrutto `switch`.
8. Niente, semplicemente non è possibile.

Esercizio 10.v)

Astrarre un mazzo di carte napoletane e creare una classe eseguibile (con metodo `main()`) che istanzi tutte le 40 carte.

Esercizio 10.z)

Partendo dalla soluzione dell'esercizio precedente:

1. Individuare i problemi di formattazione.
2. Risolvere i problemi di formattazione.

Questo esercizio, testa la vostra capacità di risolvere i problemi con inventiva, coscienza e intraprendenza. Praticamente quello che ogni programmatore dovrebbe fare sempre!

Soluzioni degli esercizi del capitolo 10

Soluzione 10.a) Tipi innestati, Vero o Falso:

1. **Vero**.
2. **Vero**.
3. **Vero**.
4. **Falso**, le classi anonime devono per forza essere istanziate.
5. **Vero**, cfr. paragrafo 10.1.2.
6. **Falso**.
7. **Vero**.
8. **Falso**, una classe anonima non ha nome.
9. **Vero**.
10. **Falso**, una classe anonima non può essere dichiarata astratta.

Soluzione 10.b) Enumerazioni, Vero o Falso:

1. **Vero**.
2. **Vero**.
3. **Falso**.

4. **Falso**.
5. **Vero**.
6. **Falso**, non è possibile utilizzare il costruttore di default se ne viene dichiarato uno esplicitamente.
7. **Vero**.
8. **Vero**.
9. **Vero**.
10. **Vero**.

Soluzione 10.c)

Il listato della enumerazione `TipoFile` è molto semplice:

```
public enum TipoFile {
    JAVA,
    C_SHARP,
    C_PLUS_PLUS,
    C;
}
```

Poi bisogna modificare la classe `File`, in modo tale da usare il tipo `TipoFile` in luogo del vecchio tipo intero:

```
public abstract class File {

    private String nome;

    private TipoFile tipo;

    public File(String nome, TipoFile tipo) {
        this.nome = nome;
        this.tipo = tipo;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public TipoFile getTipo() {
        return tipo;
    }
```

```
    public void setTipo(TipoFile tipo) {
        this.tipo = tipo;
    }
}
```

Poi bisogna modificare la classe `FileSorgente`, in modo tale da usare il tipo `TipoFile` in luogo del vecchio tipo intero:

```
public class FileSorgente extends File {

    private String contenuto;

    public FileSorgente(String nome, TipoFile tipo) {
        super(nome, tipo);
    }

    public FileSorgente(String nome, TipoFile tipo, String contenuto) {
        this(nome, tipo);
        this.contenuto = contenuto;
    }

    public void aggiungiTesto(String testo) {
        if (contenuto != null) {
            contenuto = "";
        }
        if (testo != null) {
            contenuto += testo;
        }
    }

    public void aggiungiTesto(String testo, int posizione) {
        final int length = contenuto.length();
        if (contenuto != null && testo != null && posizione > 0 &&
            posizione < length) {
            contenuto = contenuto.substring(0, posizione) + testo +
            contenuto.substring(posizione);
        }
    }

    public String getContenuto() {
        return contenuto;
    }

    public void setContenuto(String contenuto) {
        this.contenuto = contenuto;
    }

}
```

Tutto il resto può restare com'è.

Soluzione 10.d)

Il listato di `File` con l'enumerazione innestata `TipoFile` dovrebbe essere il seguente:

```
public abstract class File {

    public enum TipoFile {

        JAVA,
        C_SHARP,
        C_PLUS_PLUS,
        C;
    }

    private String nome;

    private TipoFile tipo;

    public File(String nome, TipoFile tipo) {
        this.nome = nome;
        this.tipo = tipo;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public TipoFile getTipo() {
        return tipo;
    }

    public void setTipo(TipoFile tipo) {
        this.tipo = tipo;
    }
}
```

Poi dobbiamo modificare le due classi di test per referenziare correttamente l'elemento innestato `TipoFile`:

```
public class TestFileSorgente {
    public static void main(String args[]) {
        FileSorgente fileSorgente = new FileSorgente("Test.java",
          File.TipoFile.JAVA, "public class MyClass {\n\r");
        System.out.println(fileSorgente.getContenuto());
        // Test aggiungiTesto (String) corretto
```

```
        fileSorgente.aggiungiTesto("}");
        System.out.println(fileSorgente.getContenuto());
        // Test aggiungiTesto (String,int) corretto
        fileSorgente.aggiungiTesto("//Test aggiunta testo\n\r", 23);
        System.out.println(fileSorgente.getContenuto());
        // Test aggiungiTesto (String,int) scorretto
        fileSorgente.aggiungiTesto("//Test aggiunta testo\n\r", -1);
        System.out.println(fileSorgente.getContenuto());
        // Test aggiungiTesto (String,int) scorretto
        fileSorgente.aggiungiTesto("//Test aggiunta testo\n\r", 100);
        System.out.println(fileSorgente.getContenuto());
    }
}
```

e:

```
public class TestIDE {

    public static void main(String args[]) {
        IDE ide = new JavaIDE();
        FileSorgente fileSorgente = new FileSorgente("Test.java",
          File.TipoFile.JAVA, "public class MyClass {\n\r");
        ide.modifica(fileSorgente, "}");
    }
}
```

Non è stato necessario invece modificare la classe `FileSorgente`, perché essendo sottoclasse di `File` ha ereditato l'enumerazione.

Soluzione 10.e)

Il listato della nostra enumerazione è abbastanza complesso:

```
public enum Valore {

    UN_CENTESIMO(1) {
                @Override
                public String getStringaDescrittiva() {
                    return getValore() + " centesimi di ";
                }
            },
    DUE_CENTESIMI(2),
    CINQUE_CENTESIMI(5),
    DIECI_CENTESIMI(10),
    VENTI_CENTESIMI(20),
    CINQUANTA_CENTESIMI(50),
    UN_EURO(1) {
                @Override
                public String getStringaDescrittiva() {
                    return getValore() + " ";
```

```
                }
            },
    DUE_EURO(2) {
                @Override
                public String getStringaDescrittiva() {
                    return getValore() + " ";
                }
            };

    private int valore;

    private Valore(int valore) {
        this.valore = valore;
    }

    public String getStringaDescrittiva() {
        return getValore() + " centesimi di ";
    }

    public int getValore() {
        return valore;
    }
}
```

Abbiamo definito tutti i possibili valori per una moneta (supposta di tipo Euro). Ogni elemento dell'enumerazione usa un costruttore che imposta il valore numerico della variabile `valore`. Inoltre è definito il metodo `getStringDescrittiva()` che semplificherà la gestione fatta nella classe `Moneta` originale. Tale metodo è sottoposto a override tramite una classe anonima per tre elementi dell'enumerazione. La classe `Moneta` va modificata (semplificata) come segue:

```
public class Moneta {

    public final static String VALUTA = "EURO";

    private final Valore valore;

    public Moneta(Valore valore) {
        this.valore = valore;
        System.out.println("Creata una " + getDescrizione());
    }

    public Valore getValore() {
        return valore;
    }

    public String getDescrizione() {
        String descrizione = "moneta da "
          + valore.getStringaDescrittiva() + VALUTA;
```

```
        return descrizione;
    }
}
```

Soluzione 10.f)

Il listato della classe `PortaMonete` va modificato nel seguente modo (commenti omessi):

```
public class PortaMonete {

    private static final int DIMENSIONE = 10;
    private final Moneta[] monete = new Moneta[DIMENSIONE];


    public PortaMonete(Valore... valori) {
        assert monete.length == DIMENSIONE;
        try {
            int numeroMonete = valori.length;
            for (int i = 0; i < numeroMonete; i++) {
                if (i >= DIMENSIONE) {
                    throw new PortaMonetePienoException(
                      "Sono state inserite solo le prime + "
                      + DIMENSIONE + " monete!");
                }
                monete[i] = new Moneta(valori[i]);
            }
//      } catch (PortaMonetePienoException | NullPointerException exc) {
        } catch (PortaMonetePienoException exc) {
            System.out.println(exc.getMessage());
        } catch (NullPointerException exc) {
            System.out.println("Il portamonete è stato creato vuoto");
        }
        assert monete.length == DIMENSIONE;
    }


    public void aggiungi(Moneta moneta) throws PortaMonetePienoException {
        try {
            System.out.println("Proviamo ad aggiungere una " +
                    moneta.getDescrizione());
        } catch (NullPointerException exc) {
            throw new MonetaNullException();
        }
        int indiceLibero = primoIndiceLibero();
        if (indiceLibero == -1) {
            throw new PortaMonetePienoException(
              "Portamonete pieno! La moneta "
              + moneta.getDescrizione() + " non è stata aggiunta...");
        } else {
```

```
            monete[indiceLibero] = moneta;
            System.out.println("E' stata aggiunta una " +
              moneta.getDescrizione());
        }
    }

    public Moneta preleva(Moneta moneta) throws MonetaNonTrovataException {
        try {
            System.out.println("Proviamo a prelevare una " +
              moneta.getDescrizione());
        } catch (NullPointerException exc) {
            throw new MonetaNullException();
        }
        Moneta monetaTrovata = null;
        int indiceMonetaTrovata = indiceMonetaTrovata(moneta);
        if (indiceMonetaTrovata == -1) {
            throw new MonetaNonTrovataException("Moneta non trovata!");
        } else {
            monetaTrovata = moneta;
            monete[indiceMonetaTrovata] = null;
            System.out.println("Una " + moneta.getDescrizione()
              + " prelevata");
        }
        return monetaTrovata;
    }

    public void stato() {
        System.out.println("Il portamonete contiene:");
        for (Moneta moneta : monete) {
            if (moneta == null) {
                break;
            }
            System.out.println("Una " + moneta.getDescrizione());
        }
    }

    private int primoIndiceLibero() {
        int indice = -1;
        for (int i = 0; i < 10; i++) {
            if (monete[i] == null) {
                indice = i;
                break;
            }
        }
        return indice;
    }

    private int indiceMonetaTrovata(Moneta moneta) {
        int indiceMonetaTrovata = -1;
        for (int i = 0; i < 10; i++) {
            if (monete[i] == null) {
```

```
                break;
            }
            Valore valoreMonetaNelPortaMoneta = monete[i].getValore();
            Valore valore = moneta.getValore();
            if (valore == valoreMonetaNelPortaMoneta) {
                indiceMonetaTrovata = i;
                break;
            }
        }
        return indiceMonetaTrovata;
    }
}
```

Soluzione 10.g)

Il listato della classe `TestMonete` è il seguente:

```
public class TestMonete {

    public static void main(String args[]) {

        Moneta monetaDaVentiCentesimi = new Moneta(Valore.VENTI_CENTESIMI);
        Moneta monetaDaUnCentesimo = new Moneta(Valore.UN_CENTESIMO);
        Moneta monetaDaUnEuro = new Moneta(Valore.UN_EURO);
        // Creaiamo un portamonete con 11 monete
        PortaMonete portaMoneteInsufficiente =
          new PortaMonete(Valore.DUE_CENTESIMI,
          Valore.CINQUE_CENTESIMI, Valore.UN_EURO, Valore.DIECI_CENTESIMI,
          Valore.CINQUANTA_CENTESIMI, Valore.DIECI_CENTESIMI,
          Valore.UN_EURO, Valore.DUE_EURO, Valore.DIECI_CENTESIMI,
          Valore.CINQUE_CENTESIMI, Valore.DUE_CENTESIMI);
        // Creaiamo un portamonete con 8 monete
        PortaMonete portaMonete = new PortaMonete(Valore.DUE_CENTESIMI,
          Valore.CINQUE_CENTESIMI, Valore.UN_EURO, Valore.DIECI_CENTESIMI,
          Valore.CINQUANTA_CENTESIMI, Valore.DIECI_CENTESIMI,
          Valore.UN_EURO, Valore.DUE_EURO);
        portaMonete.stato();
        try {
            // Aggiungiamo una moneta da 20 centesimi
            portaMonete.aggiungi(monetaDaVentiCentesimi);
        } catch (PortaMonetePienoException | MonetaNullException exc) {
            System.out.println(exc.getMessage());
        }
        try {
            // Aggiungiamo la decima moneta da 1 centesimo.
            portaMonete.aggiungi(monetaDaUnCentesimo);
        } catch (PortaMonetePienoException | MonetaNullException exc) {
            System.out.println(exc.getMessage());
        }
        try {
```

```
        // Aggiungiamo l'undicesima moneta (dovremmo ottenere un errore e
        // la moneta non sarà aggiunta)
            portaMonete.aggiungi(monetaDaUnEuro);
        } catch (PortaMonetePienoException | MonetaNullException exc) {
            System.out.println(exc.getMessage());
        }// Valutiamo lo stato del portamonete.
        portaMonete.stato();
        try {
            // preleviamo 20 centesimi
            portaMonete.preleva(monetaDaVentiCentesimi);
        } catch (MonetaNonTrovataException exc) {
            System.out.println(exc.getMessage());
        }
        try {
            // Aggiungiamo l'undicesima moneta (dovremmo ottenere un
            //errore e la moneta non sarà aggiunta)
            portaMonete.aggiungi(monetaDaUnEuro);
        } catch (PortaMonetePienoException | MonetaNullException exc) {
            System.out.println(exc.getMessage());
        }
        portaMonete.stato();
        //il prossimo esempio non è più riproducibile
//        try {
//
//Cerchiamo una moneta non esistente (dovremmo ottenere una stampa
//  di errore)
//
//            portaMonete.preleva(new Moneta(7));
//        } catch (MonetaNonTrovataException exc) {
//            System.out.println(exc.getMessage());
//        }
        //testiamo il passaggio di null al costruttore del porta monete
        PortaMonete portaMoneteEccezione = new PortaMonete(null);
        portaMonete.stato();
        try {
            // proviamo ad aggiungere null
            portaMonete.aggiungi(null);
        } catch (PortaMonetePienoException | MonetaNullException exc) {
            System.out.println(exc.getMessage());
        }
        try {
            //Proviamo a prelevare una moneta null
            portaMonete.preleva(null);
        } catch (MonetaNonTrovataException | MonetaNullException exc) {
            System.out.println(exc.getMessage());
        }
    }
}
```

Soluzione 10.h)

La soluzione è molto semplice utilizzando una classe anonima al volo come nel seguente listato, la difficoltà semmai è nella formattazione del codice sfruttando tabulazioni ("`\t`"):

```
public class TestPersona {

    public static void main(String args[]) {
        System.out.println(new Persona("Arjen Anthony", "Lucassen",
              "03/04/1960", "Compositore", "Olanda") {

            @Override
            public String toString() {
                String string = "Nome: \t\t\t" + getNome();
                string += "\nCognome: \t\t" + getCognome();
                string += "\nProfessione: \t\t" + getProfessione();
                string += "\nData di Nascita: \t" + getDataDiNascita();
                string += "\nIndirizzo: \t\t" + getIndirizzo();
                return string;
            }
        });
    }
}
```

Soluzione 10.i)

La risposta corretta è l'ultima. Infatti l'espressione `super.intero` implicherebbe che la superclasse della classe `Interna` abbia una variabile che si chiama `intero`, cosa che in realtà non sussiste. Segue il messaggio di errore della compilazione:

```
Esterna.java:11: error: cannot find symbol
           System.out.println(super.intero);
                                    ^
  symbol: variable intero
1 error
```

Soluzione 10.l)

La soluzione che appare più semplice è la seguente:

```
public class Esterna {
    private int intero = 5;

    public static void main(String[] args) {
        Esterna.Interna interna = new Esterna().new Interna();
        interna.metodoInterno();
    }
```

```
    private class Interna extends Esterna {
        private int intero = 10;
        protected void metodoInterno() {
            System.out.println(super.intero);
            System.out.println(this.intero);
        }
    }
}
```

Abbiamo evidenziato in grassetto le modifiche apportate.

Soluzione 10.m)

Le affermazioni corrette sono le 1, 2, 3, 4 e 5.

Soluzione 10.n)

L'affermazione corretta è la 3. Infatti il codice del metodo `main()` istanzia una classe anonima al volo, ridefinisce il metodo `metodo()` e, senza assegnare la nuova istanza ad un reference, chiama (sempre al volo) il metodo appena ridefinito.

Sarebbe stato opportuno anche annotare il metodo ridefinito con l'annotazione `Override`.

Soluzione 10.o)

Tutte le affermazioni sono vere. In particolare la numero 1 è vera, perché vale la regola numero 7 (cfr. paragrafo 10.1.2): "Una classe interna può dichiarare membri statici solo se dichiarata statica.".

Soluzione 10.p)

L'affermazione 1 è corretta. Infatti non è possibile compilare questo codice, perché non è possibile dichiarare una classe innestata all'interno di un costruttore (o di un metodo).

La numero 2 è falsa perché, mentre la prima affermazione potrebbe considerarsi corretta (se è dichiarata statica una variabile non può essere locale, e solo le variabili locali possono essere effettivamente `final`), la seconda non è vera perché

la classe innestata `Innestata`, è a sua volta dichiarata statica, e quindi potrebbe utilizzare le variabili statiche della classe `Esterna`. Per lo stesso motivo la numero 3 è anch'essa falsa. La numero 4 è ovviamente falsa perché, come abbiamo detto per l'affermazione 1, non è possibile dichiarare una classe innestata all'interno di un costruttore (o di un metodo).

Soluzione 10.q)

Le risposte corrette sono le 2, 4 e persino la 5.

Soluzione 10.r)

La risposta giusta è la 2, ovvero viene stampato `29`. Infatti il metodo `metodo()`, ereditato dall'interfaccia `InnerInterface`, è riscritto dall'enumerazione `Esercizio10R` con un'implementazione che restituisce il numero `22`. Ma per lo specifico elemento dell'enumerazione `TRE`, il metodo viene nuovamente riscritto in modo tale da restituire appunto il valore `29`. Nel metodo `main()` l'elemento `Esercizio10R.TRE` viene assegnato al reference `i` dell'interfaccia `Interface` (è legale per il polimorfismo per dati). Poi viene invocato virtualmente il metodo `metodo()` su `i`, che in realtà punta all'elemento `TRE`, che come abbiamo detto, ha ridefinito un'implementazione del metodo `metodo()`, che restituisce `29`.

Soluzione 10.s)

La risposta corretta è l'ultima (la numero 9). Infatti non è possibile dichiarare una enumerazione all'interno di una classe interna (cfr. paragrafo 10.3.2).
L'output della compilazione è il seguente:

```
Esercizio10S.java:4: error: enum declarations allowed only in static
  contexts
        public enum EnumInterna {
               ^
1 error
```

Soluzione 10.t)

Interpretando l'output della soluzione precedente:

```
Esercizio10S.java:4: error: enum declarations allowed only in static
  contexts
        public enum EnumInterna {
               ^
1 error
```

Si arriva facilmente a capire che il problema è che le classi interne (che ricordiamo essere classi innestate non statiche) non possono dichiarare enumerazioni. Il compilatore però ci dice che le dichiarazioni di enumerazioni sono permesse solo in contesti statici, così viene naturale dichiarare la classe innestata come statica in questo modo:

```
public enum Esercizio10T {

    A, B, C;

    public static class Interna {
        public enum EnumInterna {
            D, E, F;
        }
    }

    public static void main(String args[]) {
        for (Esercizio10S.Interna.EnumInterna item :
             Esercizio10S.Interna.EnumInterna.values()) {
            System.out.println(item);
        }
    }
}
```

Compilando questo codice, otterremo il seguente output:

```
Esercizio10T.java:9: error: package Esercizio10S.Interna does not exist
        for (Esercizio10S.Interna.EnumInterna item :
  Esercizio10S.Interna.EnumInterna.values()) {
  ^
D:\claudiodesio.com\Manuale\java9\Codice\capitolo_10\esercizi\10.t\
  Esercizio10T.java:9: error: package Esercizio10S.Interna does not
  exist for (Esercizio10S.Interna.EnumInterna item : Esercizio10S.
  Interna.EnumInterna.values()) {
                      ^
2 errors
```

Il che ci fa capire che il tipo `Esercizio10S.Interna.EnumInterna` non viene riconosciuto come tipo valido. La soluzione consiste nell'utilizzare il tipo corretto per il nostro contesto di visibilità: `Interna.EnumInterna` (ovvero non abbiamo specificato l'enumerazione che contiene sia il metodo `main()`, sia la classe `Interna`). Quindi il codice dovrebbe essere il seguente:

```
public enum Esercizio10T {

    A, B, C;

    public static class Interna {
```

```
        public enum EnumInterna {
            D, E, F;
        }
    }

    public static void main(String args[]) {
        for (Interna.EnumInterna item : Interna.EnumInterna.values()) {
            System.out.println(item);
        }
    }
}
```

Che verrà compilato senza errori, ed il cui output risulterà essere il seguente:

```
D
E
F
```

Come richiesto.

Soluzione 10.u)

La risposta è la numero 6, come è possibile verificare nel paragrafo 10.2.1 quando nell'esempio della videoteca avevamo creato il codice da far eseguire al volo ad un metodo con la seguente sintassi:

```
public class TestVideoteca {
    public static void main(String args[]) {
//...
        Videoteca videoteca = new Videoteca();
        System.out.println("Bei Film:");
        Film[] beiFilms = videoteca.getFilmFiltrati(new FiltroFilm() {
        @Override
            public boolean filtra(Film film) {
                return film.getMediaRecensioni() >3;
            }
    });
//...
        System.out.println("\nFilm di Fantascienza:");
        Film[] filmDiFantascienza = videoteca.getFilmFiltrati(
          new FiltroFilm() {
            @Override
            public boolean filtra(Film film) {
                return"Fantascienza".equals(film.getGenere());
            }
        });
//...
```

dove la classe `Videoteca` era così codificata:

```
public class Videoteca {
    private Film[] films;

    public Videoteca () {
        films = new Film[10];
        caricaFilms();
    }

    public void setFilms(Film[] films) {
        this.films = films;
    }

    public Film[] getFilms() {
        return films;
    }

/*  I SEGUENTI METODI SONO STATI SOSTITUITI DAL METODO DEFINITO SUBITO DOPO
    public Film[] getFilmDiFantascienza() {
        Film [] filmDiFantascienza = new Film[10];
        for (int i = 0, j= 0; i< 10;i++) {
            if ("Fantascienza".equals(films[i].getGenere())) {
                filmDiFantascienza[j] = films[i];
                j++;
            }
        }
        return filmDiFantascienza;
    }

    public Film[] getBeiFilm() {
        Film [] beiFilms = new Film[10];
        for (int i = 0, j= 0; i< 10;i++) {
            if (films[i].getMediaRecensioni() > 3) {
                beiFilms[j] = films[i];
                j++;
            }
        }
        return beiFilms;
    } */

    /* QUESTO METODO SOSTITUISCE I DUE PRECEDENTI (COMMENTATI) */
    public Film[] getFilmFiltrati(FiltroFilm filtroFilm) {
        Film [] filmFiltrati = new Film[10];
        for (int i = 0, j= 0; i< 10;i++) {
            if (filtroFilm.filtra(films[i])) {
                filmFiltrati[j] = films[i];
                j++;
            }
        }
        return filmFiltrati;
    }
```

```
    private void caricaFilms() {
        //Caricamento film...
    }
}
```

e dove l'interfaccia `FiltroFilm`, che viene ridefinita con le classi anonime nella classe `TestVideoteca`, era semplicemente la seguente:

```
public interface FiltroFilm {
    boolean filtra(Film film);
}
```

Soluzione 10.v)

Abbiamo deciso di utilizzare le enumerazioni sia per astrarre il concetto di seme che quello del numero delle carte. Un numero potrebbe essere rappresentato con un tipo primitivo come un `int`, o anche un `byte`. Ma tutto sommato per il nostro scopo l'enumerazione è parsa la scelta migliore per poter rappresentare anche testualmente le varie carte. Di seguito il codice dell'enumerazione `Numero`:

```
public enum Numero {

    UNO("Asso"),
    DUE("Due"),
    TRE("Tre"),
    QUATTRO("Quattro"),
    CINQUE("Cinque"),
    SEI("Sei"),
    SETTE("Sette"),
    OTTO("Otto"),
    NOVE("Nove"),
    DIECI("Dieci");

    String rappresentazione;

    Numero(String rappresentazione) {
        this.rappresentazione = rappresentazione;
    }

    @Override
    public String toString() {
        return rappresentazione;
    }
}
```

Si noti che abbiamo sfruttato la variabile `rappresentazione` per gestire la rappresentazione testuale degli elementi dell'enumerazione. Questa viene impostata nel

momento della definizione sfruttando l'unico costruttore fornito. Con questa variabile abbiamo potuto far stampare la stringa `Asso` in luogo di `Uno`, in modo piuttosto semplice (ovvero senza utilizzare particolari algoritmi che usano condizioni come `switch` o `if`). Se avessimo utilizzato un tipo primitivo come `int` invece avremmo dovuto gestire la situazione con un algoritmo, che implica maggiori probabilità di errore. Si noti che abbiamo anche ridefinito il metodo `toString()` ereditato dalla classe `Object` per facilitare la stampa degli elementi dell'enumerazione.
Per essere sicuri di aver scritto del codice corretto, abbiamo anche creato una classe di test, praticamente uno unit test (cfr. paragrafo G.4.1 dell'appendice G) artigianale e semplificato per testare solo la stampa degli elementi dell'enumerazione:

```
/**
 * Classe che testa l'enumerazione Numero.
 */
public class TestNumero {
    public static void main(String args[]) {
        for (Object object : Numero.values()) {
            System.out.println(object);
        }
    }
}
```

Il cui output ha verificato la correttezza del codice (secondo le nostre intenzioni):

```
Asso
Due
Tre
Quattro
Cinque
Sei
Sette
Otto
Nove
Dieci
```

Passiamo all'enumerazione che rappresenta il `Seme` di una carta:

```
public enum Seme {

    COPPE,
    BASTONI,
    DENARI,
    SPADE;

    public String toString() {
        return rendiMaiuscolo(this.name());
    }
```

```
    private String rendiMaiuscolo(String stringa) {
        //rendiamo minuscola la stringa
        String minuscolo = stringa.toLowerCase();
        //recuperiamo la prima lettera della stringa
        String iniziale = minuscolo.substring(0,1);
        //rendiamo maiuscola la prima lettera
        String inizialeMaiuscola = iniziale.toUpperCase();
        //ritorniamo la concatenazione tra la lettera maiuscola
        //e il resto della stringa minuscola
        return inizialeMaiuscola + minuscolo.substring(1);
    }
}
```

In questo caso, per scopi didattici, abbiamo modificato il modo in cui vengono rappresentati gli elementi dell'enumerazione. Abbiamo realizzato un metodo d'utilità chiamato `rendiMaiuscolo()`, che rende maiuscola la stringa passata in input. Le istruzioni di tale metodo sono commentate e facilmente comprensibili. Il metodo `toString()`, non fa altro che restituire la stringa maiuscola del nome dell'elemento dell'enumerazione. Anche in questo caso abbiamo creato una classe di test:

```
/**
 * Classe che testa l'enumerazione Seme.
 */
public class TestSeme {
    public static void main(String args[]) {
        for (Object object : Seme.values()) {
            System.out.println(object);
        }
    }
}
```

che una volta eseguita produce l'output voluto:

```
Coppe
Bastoni
Denari
Spade
```

La classe `Carta` è molta semplice:

```
public class Carta {

    private Seme seme;
    private Numero numero;

    public Carta (Numero numero, Seme seme) {
        this.numero = numero;
        this.seme = seme;
    }
```

```
    public void setNumero(Numero numero) {
        this.numero = numero;
    }

    public Numero getNumero() {
        return numero;
    }

    public void setSeme(Seme seme) {
        this.seme = seme;
    }

    public Seme getSeme() {
        return seme;
    }

    public String toString() {
        return numero + " di " + seme;
    }
}
```

La classe più complicata è indubbiamente la classe `MazzoDiCarte`, che abbiamo voluto astrarre con un array bidimensionale di 4 righe (una per ogni seme) e 10 colonne (una per ogni numero):

```
public class MazzoDiCarte {

    private Carta[][] carte;

    public MazzoDiCarte() {
        carte = new Carta[4][10];
        caricaCarte();
    }

    public void caricaCarte() {
        Seme[] semi = Seme.values();
        Numero[] numeri = Numero.values();
        int semiLength = semi.length;
        int numeriLength = numeri.length;
        for (int i = 0; i < semiLength; i++) {
            for (int j = 0; j < numeriLength; j++) {
                carte[i][j] = new Carta(numeri[j], semi[i]);
            }
        }
    }

    public String toString() {
        String string = "";
        for (int i = 0; i < carte.length; i++) {
```

```
            for (int j = 0; j < carte[i].length; j++) {
                string += carte[i][j] + ", ";
            }
            string += "\n";
        }
        return string;
    }

    public void setCarte(Carta[][] carte) {
        this.carte = carte;
    }

    public Carta[][] getCarte() {
        return carte;
    }
}
```

Si noti che il costruttore istanzia l'unica variabile d'istanza (ovvero l'array bidimensionali `carte`) e poi invoca il metodo `caricaCarte()`, che si occupa di caricare nell'array tutte le carte del mazzo con un doppio ciclo `for`. Il metodo `toString()` invece crea e restituisce la stringa rappresentativa del mazzo di carte.
Infine, la classe principale (quella con il metodo `main()`) è la seguente:

```
public class Esercizio10V {
    public static void main(String args[]) {
        MazzoDiCarte mazzoDiCarte = new MazzoDiCarte();
        System.out.println(mazzoDiCarte);
    }
}
```

Dove con un doppio ciclo abbiamo stampato le quaranta carte del mazzo.
L'output è il seguente (purtroppo la pagina è troppo stretta rispetto ad ogni riga da stampare, quindi l'output risulta un po' sfalsato):

```
Asso di Coppe, Due di Coppe, Tre di Coppe, Quattro di Coppe, Cinque di
  Coppe, Sei di Coppe, Sette di Coppe, Otto di Coppe, Nove di Coppe,
  Dieci di Coppe,

Asso di Bastoni, Due di Bastoni, Tre di Bastoni, Quattro di Bastoni,
  Cinque di Bastoni, Sei di Bastoni, Sette di Bastoni,
  Otto di Bastoni, Nove di Bastoni, Dieci di Bastoni,

Asso di Denari, Due di Denari, Tre di Denari, Quattro di Denari,
  Cinque di Denari, Sei di Denari, Sette di Denari, Otto di Denari,
  Nove di Denari, Dieci di Denari,

Asso di Spade, Due di Spade, Tre di Spade, Quattro di Spade, Cinque di
  Spade, Sei di Spade, Sette di Spade, Otto di Spade, Nove di Spade,
  Dieci di Spade,
```

Si noti anche che la formattazione è tutt'altro che perfetta. Cercheremo di risolvere il problema nel prossimo esercizio.

Soluzione 10.z)

I problemi di formattazione sono:

1. Alla fine di ogni riga viene stampato una virgola e uno spazio non necessari.
2. Alla fine della stringa vengono stampati due a capo non necessari

Risolviamo i due problemi con due controlli (in grassetto nel codice sottostante) all'interno dei due cicli del metodo, uno con un operatore ternario che aggiunge una virgola e uno spazio solo se non siamo all'ultima iterazione sui numeri, e uno con un `if` che aggiunge un "a capo" solo se non siamo all'ultima iterazione dei semi:

```
//...
    public String toString() {
        int carteLength = carte.length;
        String string = "";
        for (int i = 0; i < carteLength; i++) {
            int rigaCarteLength = carte[i].length;
            for (int j = 0; j < rigaCarteLength; j++) {
                string += carte[i][j]
                  + (j != (rigaCarteLength-1) ? ", " : "");
            }
            if (i != (carteLength-1)) {
                string += "\n";
            }
        }
        return string;
    }
//...
```

Esercizi del capitolo 11

Tipi Generici

I tipi generici rappresentano un argomento che può diventare molto complesso. Tra questi esercizi, ed in particolare tra quelli a risposta multipla che supportano la certificazione Java, ne troverete alcuni davvero complicati, perché per rispondere correttamente dovrete avere una preparazione eccellente.

Esercizio 11.a) Generics, Vero o Falso:

1. I tipi generici e i tipi parametro sono la stessa cosa.
2. Un vantaggio principale dei generics, è che permettono di evitare bachi come quelli provocati da una `ClassCastException` al runtime, individuandoli in compilazione.
3. Le collection possono essere usate anche senza specificare i tipi parametro. In tal caso si parla di raw type.
4. L'ereditarietà ignora i tipi parametro.
5. Le wildcard si usano quando non abbiamo tipi parametrici a disposizione da usare.
6. Se creiamo un tipo generico con tipo parametro `<E>`, possiamo usare lo stesso parametro anche nei metodi dichiarati nel tipo.
7. Nei metodi generici il tipo parametro viene messo come parametro di input del metodo.

8. Il seguente codice:

```
public class MyGeneric <Pippo extends Number> {
    private List<Pippo> list;
    public MyGeneric () {
        list = new ArrayList<Pippo>();
    }
    public void add(Pippo pippo) {
        list.add(pippo);
    }
    public void remove(int i) {
        list.remove(i);
    }
    public Pippo get(int i) {
        return list.get(i);
    }
    public void copy(ArrayList<?> al) {
        Iterator<?> i = al.iterator();
        while (i.hasNext()) {
            Object o = i.next();
            add(o);
        }
    }
}
```

compila senza errori.

9. Il seguente codice:

```
public <N extends Number> void print(List<N> list) {
    for (Iterator<N> i = list.iterator();
        i.hasNext( ); ) {
        System.out.println(i.next());
    }
}
```

compila senza errori.

10. Per risolvere le wildcard capture bisogna usare un metodo helper definito nella libreria standard.

11. Il seguente codice:

```
List<String> strings = new ArrayList<String>();
strings.add(new Character('A'));
```

compila senza errori.

12. Il seguente codice:

```
List<int> ints = new ArrayList<int>();
```

compila senza errori.

13. Il seguente codice:

```
List<int> ints = new ArrayList<Integer>();
```

compila senza errori.

14. Il seguente codice:

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
```

compila senza errori.

15. Il seguente codice:

```
List ints = new ArrayList<Integer>();
```

compila senza errori.

Esercizio 11.b)

Consideriamo i sorgenti creati con gli esercizi realizzati per il capitolo 6 e poi ridefiniti nel capitolo 9 e 10. Sostituire l'array `monete` definito nella classe `PortaMonete` con `ArrayList` di oggetti `Moneta`. Eventualmente modificare le altre classi in modo tale che funzioni tutto come prima.

Suggerimento: nella documentazione della classe `ArrayList` potrebbero esserci metodi utili.

Esercizio 11.c)

Creare una classe astratta `Frutta` che definisce una variabile incapsulata `peso`. Per il nostro programma un oggetto `Frutta` senza peso non ha senso. Creare le sue sottoclassi `Mela`, `Pesca`, `Arancia`. Creare una classe generica `Cesto` che astrae il concetto di cesto della frutta. Questa definisce un array list di frutti. Inoltre deve esporre un metodo `getPeso()` che ritorna il peso totale del contenuto della cesta; un metodo `aggiungi()` per aggiungere un frutto alla volta, che rilancia un'eccezione (creata appositamente) nel caso l'aggiunta del frutto che si vuole aggiungere faccia superare il limite massimo di 5 chili di peso. Ogni `Cesta` deve avere un solo tipo di frutta per volta. Implementare una soluzione con i generics. Creare una classe di test per verificare l'effettivo funzionamento del programma.

Esercizio 11.d)

Quali dei seguenti statement è compilabile (scegliere tutti gli statement corretti):

1. `HashMap hm = new HashMap();`
2. `HashMap<> hm = new HashMap<>();`
3. `HashMap<Integer, String> hm = new HashMap ()<Integer, String>;`
4. `HashMap<Double> hm = new HashMap<Double>();`
5. `HashMap<Double> hm = new HashMap<>();`

Esercizio 11.e)

Quali dei seguenti statement è compilabile (scegliere tutti gli statement corretti):

1. `ArrayList al = new ArrayList<Integer>();`
2. `ArrayList<Integer> al = new ArrayList<>();`
3. `ArrayList<String, String> al = new ArrayList<String, String>();`
4. `ArrayList al = new ArrayList<>();`
5. `ArrayList al = new ArrayList<int>();`

Esercizio 11.f)

Quali dei seguenti statement è compilabile (scegliere tutti gli statement corretti):

1. `List al = new ArrayList<HashMap<String, String>>();`
2. `Map<ArrayList<String>> hm = new ArrayList<>();`
3. `List<String, String> al = new ArrayList<HashMap<String, String>>();`
4. `List<Map<List<List<Integer>>, String>> al = new ArrayList<>();`
5. `List<Map<String, String>> al = new ArrayList<HashMap<String, String>>();`

Esercizio 11.g)

Qual è l'output delle seguenti righe di codice?

```
import java.util.*;

public class Esercizio11G {
```

```
    public static void main(String args[]) {
        List list = new ArrayList();
        list.add("1");
        list.add('2');
        list.add(3);
        Iterator iterator = list.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next());
        }
    }
}
```

1. `123`
2. `321`
3. Nessuno, il programma non si può compilare perché non è possibile aggiungere elementi diversi alla collezione.
4. `"1"'2'3`
5. Nessuno, il programma non si può compilare perché non è possibile recuperare un `Iterator` da una collezione eterogenea.
6. L'esecuzione sarà stoppata nel momento in cui si proverà a stampare il secondo valore della collezione.

Esercizio 11.h)

Qual è l'output delle seguenti righe di codice?

```
import java.util.*;

public class Esercizio11G {

   public static void main(String args[]) {
        List<Object> list = new ArrayList<>();
        list.add("1");
        list.add('2');
        list.add(3);
        Iterator<Object> iterator = list.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next());
        }
    }
}
```

1. `123`
2. `"1"'2'3`

3. Nessuno, il programma non si può compilare perché la collezione ammette solo oggetti di tipo `Object`.
4. Nessuno, il programma non si può compilare perché non è possibile recuperare un `Iterator` da una collezione eterogenea.
5. L'esecuzione sarà stoppata nel momento in cui si proverà a stampare il secondo valore della collezione.

Esercizio 11.i)

Qual è l'output delle seguenti righe di codice?

```
import java.util.*;

public class Esercizio11H {

    public static void main(String args[]) {
        List<String> list = new ArrayList<>();
        list.add("1");
        list.add(null);
        list.add('3');
        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next());
        }
    }
}
```

1. Nessuno, il programma non si può compilare perché la collezione non ammette valori `null`.
2. Nessuno, il programma non si può compilare perché la collezione non ammette valori primitivi.
3. L'esecuzione sarà stoppata nel momento in cui si proverà ad aggiungere `null` alla collezione.
4. L'esecuzione sarà stoppata nel momento in cui si proverà ad aggiungere `'3'` alla collezione.
5. `1null3`

Esercizio 11.l)

Qual è l'output delle seguenti righe di codice?

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

```
public class Esercizio11L {

    public static void main(String args[]) {
        Map<Integer, Integer> map = new HashMap<>(1);
        map.put(14, 12);
        map.put('a','b');
        map.put(07,0x0ABC);
    }
}
```

1. Il codice non compila perché non è possibile aggiungere tre coppie chiave-valore ad una collection di dimensione 1.
2. Il codice non compila perché i valori `07` e `0x0ABC` non sono valori interi.
3. Il codice non compila perché i valori `'a'` e `'b'` non sono valori interi.
4. Nessuna delle precedenti.

Esercizio 11.m)

Data la seguente dichiarazione:

```
Map<String, Float> map = new Hashmap<>(1);
```

Quali tra i seguenti statement sono validi?

1. `map.add("stringa", 1.1F);`
2. `map.add("stringa", 1.1D);`
3. `map.add("stringa", 1);`
4. Nessuna delle precedenti.

Esercizio 11.n)

Data la seguente classe:

```
public class Esercizio11N/*INSERISCI CODICE 1*/ {

    public static void main(String[] args) {
        Esercizio11N<String> g =
          new Esercizio11N/*INSERISCI CODICE 2*/();
        Esercizio11N<Object> g2 =
          new Esercizio11N/*INSERISCI CODICE 3*/();
    }
}
```

Se volessimo inserire le tre istruzioni mancanti affinché la compilazione vada a buon fine, quali tra queste opzioni risulteranno essere valide?

1. Codice 1: `<Object>`; Codice 2: `<Object>`; Codice 3: `<Object>`.
2. Codice 1: `<>`; Codice 2: `<>`; Codice 3: `<>`.
3. Codice 1: nessun codice; Codice 2: `<>`; Codice 3: `<>`.
4. Codice 1: `<T extends Object>`; Codice 2: `<String>`; Codice 3: `<Object>`.
5. Codice 1: `<? extends Object>`; Codice 2: `<String>`; Codice 3: `<Object>`.
6. Codice 1: `<T>`; Codice 2: `<String>`; Codice 3: `<>`.
7. Codice 1: `<T>`; Codice 2: `<T>`; Codice 3: `<Object>`.

Esercizio 11.o)

Data la seguente classe:

```
import java.util.*;

public class Esercizio11O {

    public static int getSize(List/*INSERISCI CODICE QUI*/ list) {
        return list.size();
    }

    public static void main(String args[]) {
        System.out.println(getSize(new ArrayList<Integer>()));
        System.out.println(getSize(
          new ArrayList<HashMap<String, List<String>>>()));
    }
}
```

Quali tra i seguenti parametri generici potrebbero sostituire il commento `/*INSERISCI CODICE QUI*/` per far sì che il codice compili correttamente, e permetta di stampare la dimensione della lista passata in input, qualsiasi sia il tipo di lista?

1. `<Object>`
2. `<>`
3. `<?>`
4. `<T extends Object>`
5. `<? extends Object>`

6. `<T>`
7. Nessun codice, il file può essere compilato così com'è.

Esercizio 11.p)

Dato il seguente codice?

```
public class Esercizio11P<T> {

    public static T[] getValues(T t) {
        return t.values();
    }

    private enum T {
        T1, T2, T3;
    }
}
```

Quale tra le seguenti affermazioni sono corrette?

1. Il codice non compila perché non è possibile chiamare l'enumerazione innestata `T` come il tipo parametro.
2. Il codice compila correttamente.
3. Il codice compila correttamente, ma mostra un warning.
4. Il codice non compila perché non è possibile dichiarare un'enumerazione come tipo parametro.
5. Il codice non compila perché un metodo statico non può accedere ad una enumerazione non statica.
6. Nessuna delle precedenti.

Esercizio 11.q)

Data la seguente classe:

```
public class Esercizio11Q/*INSERISCI CODICE QUI*/ {

    public Integer max(N n1, N n2) {
        return Integer.max(n1.intValue(), n2.intValue());
    }
}
```

Quale tra i seguenti parametri generici potrebbe sostituire il commento `/*INSERISCI CODICE QUI*/` per far sì che il codice compili correttamente:

1. `<Object>`
2. `<Number>`
3. `<?>`
4. `<T>`
5. `<N>`
6. `<N extends Number>`
7. `<T>`
8. Nessun codice, il file può essere compilato così com'è.

Esercizio 11.r)

Data la seguente classe:

```
import java.util.List;
import java.util.ArrayList;

public class Esercizio11R<T> {

    public static void main(String args[]) {
        Esercizio11R<Void> e = new Esercizio11R<>();
        ArrayList<Integer> a = new ArrayList<>();
        a.add(2);
        a.add(6);
        a.add(10);
        a.add(24);
        a.add(17);
        System.out.println("La somma degli elementi della lista è "
          + e.sumElements(a));
    }
}
```

Creare un metodo generico (cfr. paragrafo 11.3.3) che:

- prenda in input un parametro generico `L` definito dal metodo, che rappresenta una lista di interi.
- Abbia come tipo di ritorno un intero, risultato della somma di tutti gli elementi della lista di interi.

Si noti che abbiamo utilizzato il tipo `Void` come generico per l'istanza di Esercizio11R, visto che comunque la classe Esercizio11R non utilizza nemmeno il parametro.

Esercizio 11.s)

Data la seguente gerarchia:

```
public interface Tecnologia {

    void stampa();
}

public class Inkjet implements Tecnologia {

    @Override
    public void stampa() {
        System.out.println("Stampa getto d'inchiostro");
    }
}

public class Laser implements Tecnologia {

    @Override
    public void stampa() {
        System.out.println("Stampa laser");
    }
}
```

Creare una classe generica `Stampante` parametrizzata con una tecnologia, che dichiara a sua volta un metodo `stampa()`, che però delega alla propria tecnologia l'effettivo messaggio da stampare. Creare anche una classe che testa l'effettivo funzionamento del nostro codice.

Esercizio 11.t)

Data la seguente enumerazione:

```
public enum Operatore {

    SOMMA("+"), SOTTRAZIONE("-"), MOLTIPLICAZIONE("X"), DIVISIONE(":");

    String segno;

    Operatore(String segno) {
        this.segno = segno;
    }

    public String toString() {
        return segno;
    }
}
```

e la classe:

```
public class TestOperazione {
    public static void main(String args[]) {
        Operazione<Integer, Operatore> operazione =
          new Operazione<>(1, Operatore.SOMMA, 1);
        operazione.stampa();
    }
}
```

creare la classe `Operazione`, con un metodo `stampa()` che stampi semplicemente:

```
1 + 1
```

Esercizio 11.u)

Data la seguente classe:

```
import java.util.List;
import java.util.Iterator;

public class Esercizio11U {
    public static <T extends Number> void cicla(List<T> list) {
        for (Iterator<T> i = list.iterator(); i.hasNext( ); ) {
            System.out.println(i.next());
        }
    }
}
```

Quali delle seguenti affermazioni riguardo il metodo `cicla()` sono corrette?

1. È un metodo che usa una wildcard lower bounded.
2. È un metodo che usa un parametro bounded.
3. È un metodo che usa una wildcard upper bounded.
4. È un metodo generico.
5. È un metodo che usa una wildcard capture.
6. È un metodo helper.
7. È un metodo che ha un parametro covariante.

Esercizio 11.v)

Data la seguente classe:

```
import java.util.List;
```

```
public class Esercizio11V {
    protected final void modifica(List<?> list) {
        list.add(list.get(0));
    }
}
```

Quali delle seguenti affermazioni riguardo il metodo `modifica()` sono corrette?

1. È un metodo che non compila.
2. È un metodo generico.
3. È un metodo helper.
4. Ha bisogno di un metodo helper.

Esercizio 11.z)

Data la seguente classe:

```
public class Esercizio11Z {

    public static <E extends Exception> void printException(E e) {
        System.out.println(e.getMessage());
    }

    public static void main(String[] args) {
        /*INSERISCI CODICE QUI*/
    }
}
```

Quale tra i seguenti statement potrebbe sostituire il commento `/*INSERISCI CODICE QUI*/` per far sì che il codice compili correttamente:

1. `Esercizio11Z.<Exception>printException(new Exception("Exception"));`
2. `Esercizio11Z.printException(new ClassCastException("ClassCastException"));`
3. `Esercizio11Z.printException(new RuntimeException("RuntimeException"));`
4. `Esercizio11Z.printException(new Throwable("Exception"));`

Soluzioni degli esercizi del capitolo 11

Soluzione 11.a) Generics, Vero o Falso:

1. **Falso**, cfr. Paragrafo 11.1.
2. **Vero**.
3. **Vero**.
4. **Vero**.
5. **Vero**.
6. **Vero**.
7. **Falso**.
8. **Falso**, otterremo il seguente errore:

```
(Pippo) in MyGeneric<Pippo> cannot be applied to
(java.lang.Object)
add(o);
^
```

9. **Vero**.
10. **Falso**, il metodo helper bisogna crearselo da soli.
11. **Falso**.
12. **Falso**.

13. Falso.

14. Vero.

15. Vero.

Soluzione 11.b)

Il listato della classe `PortaMonete` cambierà molto:

```
import java.util.ArrayList;
import java.util.List;

/**
 * Astrae il concetto di portamonete che può contenere un numero
 * limitato di monete.
 *
 * @author Claudio De Sio Cesari
 */
public class PortaMonete {

    /**
     * Costante che rappresenta il numero massimo di monete che questo
     * portamonete può contenere.
     */
     private static final int DIMENSIONE = 10;

    /**
     * Un array list che contiene un numero limitato di monete.
     */
    private final List<Moneta> monete = new ArrayList<>(DIMENSIONE);

    /**
     * Crea un oggetto portamonete contenente monete i cui valori sono
     * specificati dal varargs valori. numero di valori che il
     * portamonete. Se il numero di elementi del varargs valori è
     * maggiore del numero massimo di monete che il portamonete
     * corrente può contenere, allora viene gestita un'eccezione
     *
     * @param valori un varargs di valori di monete.
     */
    public PortaMonete(Valore... valori) {
        assert monete.size() <= DIMENSIONE;
        try {
            int numeroMonete = valori.length;
            for (int i = 0; i < numeroMonete; i++) {
                if (i >= DIMENSIONE) {
                    throw new PortaMonetePienoException(
                      "Sono state inserite solo le prime + "
                      + DIMENSIONE + " monete!");
```

```
                }
                monete.add(new Moneta(valori[i]));
            }
        } catch (PortaMonetePienoException exc) {
            System.out.println(exc.getMessage());
        } catch (NullPointerException exc) {
            System.out.println("Il portamonete è stato creato vuoto");
        }
        assert monete.size() <= DIMENSIONE;
    }

    /**
     * Aggiunge una moneta al portamonete. Se questo è pieno la moneta
     * non è aggiunta e viene stampato un errore significativo.
     *
     * @param moneta la moneta da aggiungere.
     * @throws PortaMonetePienoException
     */
    public void aggiungi(Moneta moneta) throws PortaMonetePienoException {
        try {
            System.out.println("Proviamo ad aggiungere una "
              + moneta.getDescrizione());
        } catch (NullPointerException exc) {
            throw new MonetaNullException();
        }
        int indiceLibero = primoIndiceLibero();
        if (indiceLibero == -1) {
            throw new PortaMonetePienoException(
              "Portamonete pieno! La moneta "
              + moneta.getDescrizione() + " non è stata aggiunta...");
        } else {
            monete.set(indiceLibero, moneta);
            System.out.println("E' stata aggiunta una "
              + moneta.getDescrizione());
        }
    }

    /**
     * Esegue un prelievo della moneta specificata dal portamonete
     * corrente. Nel caso non sia presente la moneta specificata, un
     * errore significativo verrà stampato e null verrà ritornato.
     *
     * @param moneta la moneta da prelevare.
     * @return la moneta trovata o null se non trovata.
     * @throws MonetaNonTrovataException
     */
    public Moneta preleva(Moneta moneta) throws MonetaNonTrovataException {
        try {
            System.out.println("Proviamo a prelevare una "
              + moneta.getDescrizione());
        } catch (NullPointerException exc) {
```

```
            throw new MonetaNullException();
        }
        Moneta monetaTrovata = null;
        int indiceMonetaTrovata = indiceMonetaTrovata(moneta);
        if (indiceMonetaTrovata == -1) {
            throw new MonetaNonTrovataException("Moneta non trovata!");
        } else {
            monetaTrovata = moneta;
            monete.set(indiceMonetaTrovata, moneta);
            System.out.println("Una " + moneta.getDescrizione()
              + " prelevata");
        }
        return monetaTrovata;
    }

    /**
     * Stampa il contenuto del portamonete.
     */
    public void stato() {
        System.out.println("Il portamonete contiene:");
        for (Moneta moneta : monete) {
            if (moneta == null) {
                break;
            }
            System.out.println("Una " + moneta.getDescrizione());
        }
    }

    /**
     * Restituisce il primo indice libero nell'array delle monete o -1
     * se il portamonete è pieno.
     *
     * @return il primo indice libero nell'array delle monete o -1 se
     * il portamonete è pieno.
     */
    private int primoIndiceLibero() {
        int indice = monete.indexOf(null);
        return indice;
    }

    private int indiceMonetaTrovata(Moneta moneta) {
        int indiceMonetaTrovata = -1;
        final int size = monete.size();
        for (int i = 0; i < size; i++) {
            if (monete.get(i) == null) {
                break;
            }
            Valore valoreMonetaNelPortaMoneta = monete.get(i).getValore();
            Valore valore = moneta.getValore();
            if (valore == valoreMonetaNelPortaMoneta) {
                indiceMonetaTrovata = i;
```

```
                    break;
                }
            }
            return indiceMonetaTrovata;
        }
    }
```

Il costruttore ora ha delle asserzioni diverse, perché la grandezza di un `ArrayList` si misura diversamente dalla grandezza di un array, ma ha mantenuto la sua logica algoritmica ed ha cambiato solo il modo in cui viene aggiunto un elemento all'array list (metodo `add()`).
Stessa logica anche per il metodo `aggiungi()`, che ha modificato solo il modo in cui viene aggiunto un elemento con indice specificato (metodo `set()`).
Il metodo privato `primoIndiceLibero()` è stato notevolmente semplificato, grazie all'utilizzo del metodo `indexOf()` dell'array list.
Il lettore dovrebbe essere in grado ora di trovare anche le differenze nel metodo `preleva()` e nel metodo `indiceMonetaTrovata()`, visto che sono molto simili a quanto abbiamo visto per i metodi `aggiungi()` e `primoIndiceTrovato()`.
Non è necessario modificare nessun'altra classe visto che la variabile `monete` che abbiamo modificato è incapsulata.

Soluzione 11.c)

Il listato della classe `Frutta` e delle sue sottoclassi li riportiamo di seguito:

```
public abstract class Frutta {

    private final int peso;

    public Frutta(int peso) {
        this.peso = peso;
    }

    public int getPeso() {
        return peso;
    }
}

public class Mela extends Frutta {
    public Mela(int peso) {
        super(peso);
    }
}

public class Pesca extends Frutta {
    public Pesca(int peso) {
        super(peso);
```

```
    }
}

public class Arancia extends Frutta {
    public Arancia(int peso) {
        super(peso);
    }
}
```

La classe che rappresenta l'eccezione potrebbe essere la seguente:

```
public class PesoException extends Exception {
    public PesoException(String messaggio) {
        super(messaggio);
    }
}
```

La classe `Cesta`, invece potremmo codificarla così:

```
import java.util.ArrayList;

public class Cesta<F extends Frutta> {

    private ArrayList<F> frutta;

    public Cesta() {
        frutta = new ArrayList<>();
    }

    public ArrayList<F> getFrutta() {
        return frutta;
    }

    public void aggiungiFrutta(F frutto) throws PesoException {
        final int nuovoPeso = getPeso() + frutto.getPeso();
        if (nuovoPeso > 2000) {
            throw new PesoException("Troppo peso: " + nuovoPeso + " grammi!");
        }
        frutta.add(frutto);
        System.out.println(frutto.getClass().getName() + " da "
          + frutto.getPeso() + " grammi aggiunta alla cesta");
    }

    public int getPeso() {
        int peso = 0;
        for (F frutto : frutta) {
            peso += frutto.getPeso();
        }
        return peso;
    }
}
```

Si noti che con il tipo parametro possiamo usare una qualsiasi sottoclasse di `Frutta`. Il tipo `F`, viene usato poi nei metodi della classe e sarà sostituito durante l'esecuzione con una sottoclasse di `Frutta`.
Infine possiamo codificare una semplice classe di test nella seguente maniera:

```
public class TestFrutta {
    public static void main(String args[]) {
        Cesta<Mela> cestaDiMele = new Cesta<>();
        for (int i = 0; i <= 20; i++) {
            try {
                Mela mela = new Mela(100);
                cestaDiMele.aggiungiFrutta(mela);
            } catch (PesoException exc) {
                System.out.println(exc.getMessage());
            }
        }
    }
}
```

Segue l'output dell'esecuzione della classe `TestFrutta`:

```
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Mela da 100 grammi aggiunta alla cesta
Troppo peso: 2100 grammi!
```

Soluzione 11.d)

La risposta corretta è la numero 1. La numero 2 è errata perché l'operatore diamond si utilizza sull'istanza non sul reference. La numero 3 è errata per l'ordine errato delle parentesi tonde e i parametri generici dell'istanza (invertendo l'ordine

sarebbe stato uno statement compilabile). Le ultime due sono errate perché un `HashMap` ha sempre due parametri.

Soluzione 11.e)

Le risposte corrette sono la numero 1, la numero 2 e la numero 4. La numero 4 in particolare è un caso particolare perché si tratta di un raw type che usa l'operatore diamond, che in questo particolare caso è praticamente opzionale. La numero 3 è errata perché un `ArrayList` ha sempre un unico parametro generico.

Soluzione 11.f)

Le risposte corrette sono la numero 1 e la numero 4. La numero 2 è scorretta perché una mappa ha sempre due parametri. La numero 3 non compila perché non c'è coincidenza tra i parametri del reference (due stringhe, e questo è già un errore perché una lista ha sempre un solo parametro) e l'istanza (un `HashMap`). La numero 5 è errata perché i parametri generici tra reference e istanza non coincidono. Infatti `Map` non coincide con `HashMap`.
Ecco infatti l'output dell'esecuzione:

```
jshell> List<Map<String, String>> al =
  new ArrayList<HashMap<String, String>>() ;
|  Error:
|  incompatible types: java.util.ArrayList<
  java.util.HashMap<java.lang.String,java.lang.String>> cannot be
  converted to java.util.List<java.util.Map<
  java.lang.String,java.lang.String>>
|  List<Map<String, String>> al =
  new ArrayList<HashMap<String, String>>();
| ^-------------------------------------^
```

Soluzione 11.g)

La risposta corretta è la 1. Infatti non avendo specificato il parametro generico (si parla in questi casi di raw type), sarà possibile aggiungere elementi di ogni tipo alla collezione. Si noti che aggiungendo parametri primitivi, questi vengono automaticamente promossi ai relativi oggetti wrapper (in questo caso `'2'` viene promossa a `Character`, mentre `3` ad `Integer`). Segue l'output dell'esecuzione:

```
123
```

Soluzione 11.h)

La risposta corretta è la 1. Segue l'output dell'esecuzione:

```
123
```

Soluzione 11.i)

La risposta corretta è la 2. Non è possibile aggiungere un tipo primitivo (né altri tipi complessi diversi da `String`) ad una collezione che è parametrizzata con un tipo `String`. Infatti l'output della compilazione è il seguente:

```
Esercizio11I.java:8: error: no suitable method found for add(char)
        list.add('3');
             ^
    method Collection.add(String) is not applicable
      (argument mismatch; char cannot be converted to String)
    method List.add(String) is not applicable
      (argument mismatch; char cannot be converted to String)
Note: Some messages have been simplified; recompile with
  -Xdiags:verbose to get full output
1 error
```

Soluzione 11.l)

L'unica affermazione corretta è la numero 3.
L'affermazione numero 1 non è valida perché le collezioni sono ridimensionabili di default, e il valore `1` passato al costruttore dell'`HashMap`, rappresenta solo la dimensione iniziale della collection.
Anche l'affermazione numero 2 non è corretta. Infatti i valori `07` e `0x0ABC` sono valori interi e quindi con l'autoboxing sono correttamente "wrappati" all'interno di oggetti `Integer`.
Invece i valori `'a'` e `'b'` di cui si parla nell'affermazione 3 sono caratteri e sono "wrappati" all'interno di oggetti `Character`.
Segue l'output della compilazione della classe `Esercizio11L`:

```
Esercizio11L.java:7: error: incompatible types: char cannot be
  converted to Integer
        map.put('a','b');
                ^
Note: Some messages have been simplified; recompile with
  -Xdiags:verbose to get full output
1 error
```

Soluzione 11.m)

La risposta è l'ultima, visto che per aggiungere coppie chiave-valore ad una mappa, si usa il metodo `put()`, non il metodo `add()` (che è un metodo dichiarato nelle liste).

Soluzione 11.n)

Le opzioni corrette sono la 4 e la 6.
L'opzione numero 1 è errata perché non si può assegnare un reference con tipo generico `String`, ad un'istanza con tipo generico `Object`. Infatti, compilando il file si ottiene il seguente output:

```
Esercizio11N.java:2: error: incompatible types: Esercizio11N<Object>
  cannot be converted to Esercizio11N<String>
    Esercizio11N<String> g = new Esercizio11N<Object>();
                             ^
  where Object is a type-variable:
    Object extends java.lang.Object declared in class Esercizio11N
1 error
```

La numero 2 fallirà alla prima riga, perché non è possibile utilizzare un operatore diamond come tipo generico di una classe. La numero 3 fallirà anch'essa in compilazione perché non abbiamo dichiarato un tipo generico per la classe. Segue l'output che riporta gli errori derivanti dal processo di compilazione:

```
Esercizio11N.java:2: error: type Esercizio11N does not take parameters
    Esercizio11N<String> g = new Esercizio11N<>();
                 ^
Esercizio11N.java:3: error: type Esercizio11N does not take parameters
    Esercizio11N<Object> g2 = new Esercizio11N<>();
                 ^
Esercizio11N.java:2: error: cannot infer type arguments for
  Esercizio11N
    Esercizio11N<String> g = new Esercizio11N<>();
                                              ^
  reason: cannot use '<>' with non-generic class Esercizio11N
Esercizio11N.java:3: error: cannot infer type arguments for
  Esercizio11N
    Esercizio11N<Object> g2 = new Esercizio11N<>();
                                               ^
  reason: cannot use '<>' with non-generic class Esercizio11N
4 errors
```

La numero 5 è anch'essa errata. Infatti le wildcard si utilizzano con i parametri di metodi. Segue l'output della compilazione:

```
Esercizio11N.java:1: error: <identifier> expected
public class Esercizio11N<? extends Object> {
                          ^
1 error
```

La numero 7 infine, produce il seguente output in compilazione:

```
Esercizio11N.java:2: error: incompatible types: Esercizio11N<T>
  cannot be converted to Esercizio11N<String>
    Esercizio11N<String> g = new Esercizio11N<T>();
                             ^
  where T is a type-variable:
    T extends Object declared in class Esercizio11N
1 error
```

dove veniamo avvisati che `Esercizio11M<T>`, non può essere convertito a `Esercizio11N<String>`.

Soluzione 11.o)

Le opzioni corrette sono la 3 e la 5, che sono sostanzialmente equivalenti, ma anche la 7 dove utilizziamo un raw type. Infatti compilando utilizzando l'opzione 7 otterremo un warning:

```
Esercizio11O.java:4: warning: [rawtypes] found raw type: List
    public static int getSize(List/*INSERISCI CODICE QUI*/ list) {
                              ^
  missing type arguments for generic class List<E>
  where E is a type-variable:
    E extends Object declared in interface List
1 warning
```

L'opzione 1 è errata, perché il parametro `Object` non ammette altri tipi parametri al di fuori di `Object`. Infatti ecco l'output:

```
Esercizio11O.java:9: error: incompatible types: ArrayList<Integer>
  cannot be converted to List<Object>
        System.out.println(getSize(new ArrayList<Integer>()));
                                   ^
Esercizio11O.java:10: error: incompatible types:
  ArrayList<HashMap<String,List<String>>> cannot be converted to
  List<Object>
        System.out.println(getSize(new ArrayList<
  HashMap<String, List<String>>>()));
                                   ^
Note: Some messages have been simplified; recompile with
  -Xdiags:verbose to get full output
2 errors
```

L'opzione 2 non ha senso (sintassi errata). Infatti ecco l'output:

```
Esercizio11O.java:4: error: illegal start of type
    public static int getSize(List<> list) {
                              ^
```

```
1 error
```

L'opzione 3 e l'opzione 6 sono entrambe errate per lo stesso motivo. Il tipo `T` non è definito da nessuna parte. Segue l'output con l'opzione 3:

```
Esercizio11O.java:4: error: cannot find symbol
    public static int getSize(List<T> list) {
                                    ^
  symbol:   class T
  location: class Esercizio11O
1 error
```

Che è praticamente identico a quello dove si utilizza l'opzione 6:

```
Esercizio11O.java:4: error: > expected
    public static int getSize(List<T extends Object> list) {
                                      ^
1 error
```

Soluzione 11.p)

L'affermazione corretta è la numero 3. Infatti l'output è il seguente:

```
Esercizio11P.java:4: warning: [static] static method should be
  qualified by type name, T, instead of by an expression
        return t.values();
                 ^
1 warning
```

Ovvero il compilatore ci sta avvertendo che il metodo `values()` è statico e quindi si dovrebbe chiamare utilizzando il nome dell'enumerazione, e non su un suo elemento.
Per il resto delle risposte è facile capire perché non siano corrette. In particolare per la numero 1 la lettera `T` è proprio l'identificatore dell'enumerazione, quindi la domanda è posta in maniera completamente errata.

Soluzione 11.q)

L'opzione corretta è la numero 6. Laddove non c'è il bounded parameter di tipo `Number`, otterremo un errore in compilazione come il seguente:

```
Esercizio11Q.java:4: error: cannot find symbol
        return Integer.max(n1.intValue(), n2.intValue());
                                  ^
  symbol:   method intValue()
```

```
  location: variable n1 of type N
  where N is a type-variable:
    N extends Object declared in class Esercizio11Q
Esercizio11Q.java:4: error: cannot find symbol
        return Integer.max(n1.intValue(), n2.intValue());
                                          ^
  symbol:   method intValue()
  location: variable n2 of type N
  where N is a type-variable:
    N extends Object declared in class Esercizio11Q
2 errors
```

Soluzione 11.r)

La soluzione potrebbe essere la seguente (il metodo richiesto è evidenziato in grassetto):

```
import java.util.List;
import java.util.ArrayList;

public class Esercizio11R<T> {

    private <L extends List<Integer>> Integer sumElements(L list) {
        int size = list.size();
        int result = 0;
        for (int i = 0; i < size; i++) {
            Integer item = list.get(i);
            result += item;
        }
        return result;
    }

    public static void main(String args[]) {
        Esercizio11R<Void> e = new Esercizio11R<>();
        ArrayList<Integer> a = new ArrayList<>();
        a.add(2);
        a.add(6);
        a.add(10);
        a.add(24);
        a.add(17);
        System.out.println("La somma degli elementi della lista è "
          + e.sumElements(a));
    }
}
```

Soluzione 11.s)

Una possibile soluzione è rappresentata dal seguente codice, che dichiara un classico bounded parameter, e, come richiesto, dichiara il metodo `stampa()` che de-

lega il suo reale funzionamento all'implementazione del metodo `stampa()` della tecnologia:

```
public class Stampante<T extends Tecnologia> {

    private T tecnologia;

    public Stampante(T tecnologia) {
        this.tecnologia = tecnologia;
    }

    public void stampa() {
        tecnologia.stampa();
    }
}
```

L'effettivo funzionamento può essere testato dalla seguente classe di test:

```
public class TestStampante {

    public static void main(String args[]) {
        Stampante<Laser> stampante = new Stampante<>(new Laser());
        stampante.stampa();
        Stampante<Inkjet> stampante2 = new Stampante<>(new Inkjet());
        stampante2.stampa();
    }
}
```

che produce il seguente output:

```
Stampa laser
Stampa getto d'inchiostro
```

Soluzione 11.t)

La classe `Operazione` potrebbe essere la seguente:

```
public class Operazione<Integer, O extends Operatore> {

    private Integer operando1;
    private O operatore;
    private Integer operando2;

    public Operazione(Integer operando1, O operatore, Integer operando2) {
        this.operando1 = operando1;
        this.operatore = operatore;
        this.operando2 = operando2;
    }
```

```
    public void stampa() {
        System.out.println(operando1 + " " + operatore + " " + operando2);
    }
}
```

Soluzione 11.u)

Le affermazioni corrette sono la 3 e la 4.

Soluzione 11.v)

Le affermazioni corrette sono la 1 e la 4. In particolare bisognerebbe creare un metodo helper come il seguente in grassetto:

```
import java.util.List;

public class Soluzione11V {
    protected final void modifica(List<?> list) {
        helperMethod(list);
    }

    private <T> void helperMethod(List<T> list) {
        list.add(list.get(0));
    }
}
```

Soluzione 11.z)

Gli statement corretti sono i primi 3, solo l'uso dello statement 4 causerebbe un errore in compilazione, segue l'output:

```
Esercizio11Z.java:6: error: method printException in class
  Esercizio11Z cannot be applied to given types;
        /*INSERISCI CODICE QUI*/Esercizio11Z.printException(
  new Throwable("Exception"));
                                              ^
  required: E
  found: Throwable
  reason: inference variable E has incompatible bounds
    upper bounds: Exception
    lower bounds: Throwable
  where E is a type-variable:
    E extends Exception declared in method <E>printException(E)
1 error
```

Nel caso della risposta numero 1, abbiamo usato una sintassi che si utilizza raramente, e di cui abbiamo solo accennato alla fine del paragrafo 11.3.3 relativamente ad un costruttore.

Esercizi del capitolo 12

La libreria indispensabile: il package `java.lang`

Come per tutti gli esercizi di questo libro, è possibile consultare la documentazione della libreria standard per trovare le soluzioni. Questo vale soprattutto nei capitoli come questi dedicati proprio alle librerie.

Esercizio 12.a) Autoboxing, autounboxing e `java.lang`, *Vero o Falso:*

1. Il seguente codice compila senza errori:

```
char c = new String("Pippo");
```

2. Il seguente codice compila senza errori:

```
int c = new Integer(1) + 1 + new Character('a');
```

3. Le regole dell'overload non cambiano con l'introduzione dell'autoboxing e dell'autounboxing.
4. Le istanze della classe `Integer` sono immutabili, e quindi non è possibile mutare il loro stato interno una volta istanziate.
5. La classe `Runtime` dipende strettamente dal sistema operativo su cui esegue.
6. La classe `Class` permette di leggere i membri di una classe (ma anche le superclassi ed altre informazioni) partendo semplicemente dal nome della classe grazie al metodo `forName()`.

7. Tramite la classe `Class` è possibile istanziare oggetti di una certa classe conoscendone solo il nome.

8. È possibile dalla versione 1.4 di Java sommare un tipo primitivo e un oggetto della relativa classe wrapper, come nel seguente esempio:

```
Integer a = new Integer(30);
int b = 1;
int c = a + b;
```

9. La clonazione di oggetti richiede obbligatoriamente una chiamata al metodo `clone()` di `Object`.

10. La classe `Math` non si può istanziare perché dichiarata `abstract`.

Esercizio 12.b)

Consideriamo la classe `PortaMonete` rifinita negli esercizi del capitolo 10, e concentriamoci sul metodo privato `indiceMonetaTrovata()`, che abbiamo definito nel seguente modo:

```
private int indiceMonetaTrovata(Moneta moneta) {
    int indiceMonetaTrovata = -1;
    final int size = monete.size();
    for (int i = 0; i < size; i++) {
        if (monete.get(i) == null) {
            break;
        }
        Valore valoreMonetaNelPortaMoneta = monete.get(i).getValore();
        Valore valore = moneta.getValore();
        if (valore == valoreMonetaNelPortaMoneta) {
            indiceMonetaTrovata = i;
            break;
        }
    }
    return indiceMonetaTrovata;
}
```

Creare un metodo `equals()` nella classe `Moneta`, in modo tale da semplificare la ricerca in questo metodo.

Esercizio 12.c)

Creare un blocco statico (cfr. paragrafo 6.8.3) all'interno della classe `Moneta`, che abbiamo rifinito negli esercizi del capitolo 10, e facciamogli stampare una frase qualsiasi. Come è possibile usare la reflection per far eseguire il blocco?

Esercizio 12.d)

Creare una classe `ReflectionInterattiva` che contiene un metodo `main()` il quale stampa i metodi di una classe specificata al runtime tramite l'utilizzo di una classe `Scanner` (già incontrata più volte in precedenza, come nell'esercizio 4.m nella classe `ProgrammaInterattivo`). Deve essere possibile specificare una classe, battere il tasto "Invio" e il programma deve stampare le firme di tutti i metodi della classe specificata.

Esercizio 12.e)

Creare una classe `CompilatoreInterattivo` che contiene un metodo `main()` il quale compila i file specificati specificata al runtime tramite l'utilizzo di una classe `Scanner` (già incontrata in più esercizi precedentemente). Deve essere possibile specificare una classe, battere il tasto "Invio" e il programma deve compilare la classe specificata.

Questo programma dovrebbe essere eseguito su riga di comando o tramite EJE, ma non tramite Eclipse o Netbeans. Infatti questi IDE compilano automaticamente i file e quindi non si riuscirebbe a capire bene se il file viene compilato correttamente dal nostro programma o dall'IDE.

Esercizio 12.f)

Creare una classe che, dato un array di interi, li ordina dal più alto al più basso.

Esercizio 12.g)

Creare una classe `WrapperComparable` che abbia le seguenti caratteristiche.

- Abbia come costante incapsulata un `Integer` che deve essere sempre valorizzato.
- Abbia un metodo `toString()` che restituisca l'intero.
- Definisca un modo di ordinamento che vada dall'intero incapsulato più alto al più basso.

Infine creare una classe che testi che l'ordinamento funzioni come ci si aspetta.

Esercizio 12.h)

Qual è l'output della seguente classe?

```
public class Esercizio12H {
    public static void main(String args[]) {
        String stringa1 ="Claudio";
        String stringa2 = new String(stringa1);
        System.out.println(stringa2 == stringa1);
        System.out.println(stringa2.equals(stringa1));
        System.out.println("Claudio".equals(stringa1));
        System.out.println("Claudio" == stringa1);
        System.out.println("Claudio" == stringa2);
    }
}
```

Esercizio 12.i)

Quali delle seguenti affermazioni sono corrette?

- ❑ Un oggetto immutabile non può essere modificato.
- ❑ Le stringhe sono oggetti immutabili.
- ❑ Le istanze delle classi wrapper sono oggetti immutabili.
- ❑ Un oggetto immutabile può essere puntato da più reference.
- ❑ Un oggetto immutabile non permette di modificare il suo stato interno.
- ❑ Un oggetto immutabile non permette di modificare il suo stato esterno.

Esercizio 12.l)

Quali tra le seguenti affermazioni sono corrette rispetto all'argomento delle compact strings?

- ❑ È possibile specificare quali stringhe devono essere rese compatte.
- ❑ Una stringa non compatta utilizza 16 bit.
- ❑ Una stringa compatta utilizza solo 8 bit.
- ❑ Un programma, per rendere tutte le stringhe compatte, è necessario compilarlo utilizzando l'opzione `-XX:-CompactStrings`.
- ❑ Utilizzando le compact strings le performance del programma saranno sempre incrementate del 50%.

Esercizio 12.m)

Qual è l'output della seguente classe?

```
public class Esercizio12M {
    public static void main(String args[]) {
        String stringa = "*** Java ***";
        stringa.toUpperCase();
        stringa.trim();
        stringa.substring(3, 8);
        stringa.trim();
        stringa.concat(String.format("Stringa = %n", stringa.length()));
        stringa += "!";
        System.out.println(stringa.length);
    }
}
```

1. `12`
2. `13`
3. `11`
4. `10`
5. `23`
6. `22`
7. `24`
8. Nessun output, sarà lanciata un'eccezione al runtime.
9. Nessun output, la classe non è compilabile.

Esercizio 12.n)

Qual è l'output della seguente classe?

```
public class Esercizio12N {
    public static void main(String args[]) {
        String stringa = "Java";
        stringa = stringa.concat(" ");
        stringa += 9;
        String risultato = "";
        if (stringa.intern() == "Java 9") {
            risultato += "intern()";
        }
        if (stringa == "Java 9") {
            risultato += "==";
        }
        if (stringa.equals("Java 9")) {
```

```
            risultato += "equals()";
        }
        System.out.println(risultato);
    }
}
```

1. `Java 9`
2. `intern()`
3. `intern()==`
4. `intern()equals()`
5. `null`
6. `intern()==equals()`
7. Una stringa vuota.
8. Nessun output, sarà lanciata un'eccezione al runtime.
9. Nessun output, la classe non è compilabile.

Esercizio 12.o)

Qual è l'output della seguente classe?

```
public class Esercizio12O {
    public static void main(String args[]) {
       String stringa1 = "123789";
       String stringa2 = stringa1.concat(System.lineSeparator());
       char [] array1 = stringa2.toCharArray();
       char [] array2 = {'4', '5', '6'};
       System.arraycopy(array2, 0, array1, 3, 3);
       System.out.println(array1);
       System.exit(0);
    }
}
```

Esercizio 12.p)

Scrivere una classe che rappresenta un testo di tipo RTF (Rich Text Format), ovvero un testo a cui è possibile modificare per esempio il carattere, il colore di sfondo, l'interlinea, la sottolineatura e così via (scegliete voi cosa deve definire). Rendere tale classe clonabile e creare un programma di test che verifichi l'effettivo funzionamento.

Esercizio 12.q)

Partendo dall'esercizio 6.z, creare una classe con metodo `main()` che legge username e password come properties di sistema. Testare la classe utilizzando le giuste opzioni da riga di comando (specificarle).

Esercizio 12.r)

Quali delle seguenti affermazioni sono corrette riguardo il garbage collection.

1. La JVM implementa solo due algoritmi per deallocare la memoria: G1GC e ParallelGC.
2. Con ParallelGC gli interventi sulla memoria sono più numerosi e intensi, e questo rende l'algoritmo meno efficiente rispetto a G1GC.
3. Era possibile utilizzare G1GC già dalla versione 7 di Java.
4. La "finalizzazione" consiste nell'eliminare gli oggetti non più utilizzati dall'applicazione.
5. La "finalizzazione" può essere invocata mediante la chiamata del metodo `runFinalization()` della classe `Object`.
6. Per utilizzare ParallelGC è necessario eseguire l'applicazione specificando l'opzione `-XX:+UseParallelGC`.

Esercizio 12.s)

Esplorando la documentazione ufficiale, o curiosando su Internet sull'argomento reflection, si può scoprire che sino a Java 8, il metodo `setAccessible()` chiamato su un oggetto `Field` (che ricordiamo astrae il concetto di variabile) o un oggetto `Method` (che astrae il concetto di metodo), permetteva alla reflection di accedere ai membri privati di una classe, violandone di fatto l'incapsulamento. Con l'introduzione dei moduli in Java 9, questo non è più possibile. Quindi, data la seguente classe:

```
public class ClasseConMembriPrivati {

    private String variabilePrivata =
      "Questa variabile è privata e non si tocca!";

    private String metodoPrivato() {
        return "Questo metodo è privato e non si tocca!";
    }
}
```

Consultando la documentazione ufficiale, create un'altra classe che con la reflection:

- provi a modificare il valore della variabile privata `variabilePrivata`;
- provi ad invocare il metodo privato `metodoPrivato()`.

Esercizio 12.t)

Data la seguente classe:

```
public class Esercizio12T {

    public static void main(String args[]) {
        int raggio = 7;
        /*INSERISCI CODICE QUI*/
        System.out.println("L'area della circonferenza di raggio 7 è "
          + area);
    }
}
```

definire la riga che deve sostituire il commento `/*INSERISCI CODICE QUI*/` per ottenere il corretto calcolo dell'area.

Ricordiamo che l'area di una circonferenza viene calcolata come:

pi greco per raggio al quadrato, ovvero se chiamiamo il raggio `r`, e `A` l'area della circonferenza, e indichiamo il numero pi greco con `π`, avremo:

$A = \pi r^2$

Consultare la documentazione ufficiale della classe `Math` prima di scrivere il codice.

Esercizio 12.u)

Qual è l'output della seguente classe?

```
public class Eserczio12U {

    public static void main(String args[]) {
        double e = Math.E;
```

```
        Math.floor(e);
        boolean b = check(e, 2.0);
        System.out.println(b);
    }

    public static Boolean check(Double a, Double b) {
        Boolean equals = null;
        if (a.equals(b)) {
            equals = true;
        }
        return equals;
    }
}
```

1. `true`
2. `false`
3. `null`
4. `2.0`
5. `2.718281828459045`
6. Nessun output, sarà lanciata un'eccezione al runtime.
7. Nessun output, la classe non è compilabile.

Esercizio 12.v)

Qual è l'output della seguente classe?

```
public class Esercizio12V {
    public static void main(String args[]) {
        Esercizio12V e = new Esercizio12V();
        e.metodo(128);
    }

    public void metodo(Integer numero) {
        System.out.println("Integer " + numero);
    }

    public void metodo(long numero) {
        System.out.println("long " + numero);
    }

    public void metodo(byte numero) {
        System.out.println("byte " + numero);
    }

    public void metodo(Byte numero) {
        System.out.println("Byte " + numero);
```

```
    }

    public void metodo(short numero) {
        System.out.println("short " + numero);
    }

    public void metodo(Double numero) {
        System.out.println("Double " + numero);
    }

    public void metodo(double numero) {
        System.out.println("double " + numero);
    }
}
```

Esercizio 12.z)

Creare un programma che simuli il gioco noto come "sasso-carta-forbici", noto anche come "morra cinese". Le istruzioni del gioco potete trovarle a questo link: https://it.wikipedia.org/wiki/Morra_cinese
Eseguito il programma l'utente deve specificare se scegliere sasso, carta o forbice, e contemporaneamente il programma deve fare la sua scelta (casualmente).

Si cerchi di utilizzare tutti i concetti appresi sinora per creare quest'applicazione (comprese le enumerazioni). Ricordarsi che è difficile fare delle scelte, ma con i metodi di analisi che abbiamo visto in precedenza possiamo procedere con più rigore e sicurezza.

Soluzioni degli esercizi del capitolo 12

Soluzione 12.a) Autoboxing, autounboxing e `java.lang`, *Vero o Falso:*

1. **Falso**.
2. **Vero**.
3. **Vero**.
4. **Vero**.
5. **Vero**.
6. **Vero**.
7. **Vero**.
8. **Falso**, dalla versione 1.5.
9. **Falso**.
10. **Falso**, non si può istanziare perché ha un costruttore privato ed è dichiarata `final` per non poter essere estesa.

Soluzione 12.b)

Il listato del metodo `equals()` l'abbiamo generato (insieme al metodo `hashcode()` che riportiamo per completezza anche se non necessario ai fini dell'esercizio) tramite Netbeans (abbiamo anche utilizzato la classe `java.util.Objects` che spiegheremo più avanti nel libro):

```
@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Moneta other = (Moneta) obj;
    if (this.valore != other.valore) {
        return false;
    }
    return true;
}

@Override
public int hashCode() {
    int hash = 7;
    hash = 97 * hash + Objects.hashCode(this.valore);
    return hash;
}
```

Quindi il metodo `indiceMonetaTrovata()` si può semplificare nel seguente modo:

```
private int indiceMonetaTrovata(Moneta moneta) {
    int indiceMonetaTrovata = -1;
    final int size = monete.size();
    for (int i = 0; i < size; i++) {
        if (monete.get(i) == null) {
            break;
        }
        if (monete.get(i).equals(moneta)) {
            indiceMonetaTrovata = i;
            break;
        }
    }
    return indiceMonetaTrovata;
}
```

Soluzione 12.c)

La reflection non è necessaria. Infatti, supponiamo di aver definito il blocco statico seguente nella classe `Moneta`:

```
static {
    System.out.println("Caricata la classe Moneta con valuta = " + VALUTA);
}
```

Basterebbe semplicemente definire una variabile della classe `Moneta` come segue:

```
public class TestReflection {
    public static void main(String args[]) {
        Moneta moneta = new Moneta(Valore.CINQUANTA_CENTESIMI);
    }
}
```

Il codice precedente produrrebbe il seguente output:

```
Caricata la classe Moneta con valuta = EURO
Creata una moneta da 50 centesimi di EURO
```

Tuttavia, se volessimo usare la reflection non sarà sufficiente il seguente codice:

```
        Class<Moneta> classMoneta = Moneta.class;
        try {
            classMoneta.newInstance();
        } catch (InstantiationException | IllegalAccessException ex) {
            ex.printStackTrace();
        }
```

Che infatti produrrà il seguente output:

```
java.lang.InstantiationException: Moneta
      at java.lang.Class.newInstance(Class.java:418)
      at TestReflection.main(TestReflection.java:10)
Caused by: java.lang.NoSuchMethodException: Moneta.<init>()
      at java.lang.Class.getConstructor0(Class.java:2971)
      at java.lang.Class.newInstance(Class.java:403)
      ... 1 more
```

Questo succede in quanto con il metodo `newInstance()` viene chiamato il costruttore senza parametri, che però non esiste!
La soluzione è chiamare il costruttore corretto con il seguente codice:

```
try {
    Constructor<Moneta> costruttore =
      classMoneta.getConstructor(Valore.class);
    costruttore.newInstance(Valore.CINQUANTA_CENTESIMI);
} catch (NoSuchMethodException | SecurityException |
         InstantiationException | IllegalAccessException |
         IllegalArgumentException | InvocationTargetException ex) {
    ex.printStackTrace();
}
```

Soluzione 12.d)

Il listato della classe `ReflectionInterattiva` potrebbe essere il seguente:

```
import java.lang.reflect.Method;
import java.util.Scanner;

public class ReflectionInterattiva {

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        String stringa = "";
        System.out.println("Digita il nome di una classe presente nella"
          + " cartella corrente e batti enter, oppure scrivi \"fine\""
          + " per terminare il programma");
        while (!(stringa = scanner.next()).equals("fine")) {
            System.out.println("Hai digitato "
              + stringa.toUpperCase() + "!");
            try {
                stampaMetodi(stringa);
            } catch (ClassNotFoundException ex) {
                ex.printStackTrace();
            }
        }
        System.out.println("Fine programma!");
    }

    private static void stampaMetodi(String stringa) throws
      ClassNotFoundException {
        Class objectClass = Class.forName(stringa);
        Method[] methods = objectClass.getDeclaredMethods();
        for (Method method : methods) {
            System.out.println(method);
        }
    }
}
```

Si noti che abbiamo utilizzato il metodo `getDeclaredMethods()` invece di `getMethods()`, perché quest'ultimo avrebbe anche stampato i metodi ereditati dalle superclassi (nel caso si specifichi la classe `Moneta` i metodi di `Object`).
L'output del precedente listato è il seguente:

```
Digita il nome di una classe presente nella cartella corrente e batti
 enter, oppure scrivi "fine" per terminare il programma
Moneta
Hai digitato MONETA!
Caricata la classe Moneta con valuta = EURO
public boolean Moneta.equals(java.lang.Object)
public int Moneta.hashCode()
public java.lang.String Moneta.getDescrizione()
public Valore Moneta.getValore()
fine
Fine programma!
```

Soluzione 12.e)

Il listato della classe `CompilatoreInterattivo` potrebbe essere il seguente:

```
import java.util.Scanner;

public class CompilatoreInterattivo {

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        String stringa = "";
        System.out.println("Digita il nome di un file java presente nella "
          + " cartella corrente e batti enter, oppure scrivi \"fine\" "
          + "per terminare il programma");
        while (!(stringa = scanner.next()).equals("fine")) {
            System.out.println("Hai digitato "
              + stringa.toUpperCase() + "!");
            try {
                compilaClasse(stringa);
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
        System.out.println("Fine programma!");
    }

    private static void compilaClasse(String stringa) throws Exception {
        Runtime runtime = Runtime.getRuntime();
        Process process = runtime.exec("javac " + stringa);
        final int exitValue = process.waitFor();
        System.out.println(exitValue == 0 ? stringa + " compilato!" :
          "Impossibile compilare " + stringa);
    }
}
```

Si noti che avremmo potuto anche utilizzare la Compiler API, con un codice semplice come il seguente:

```
JavaCompilerTool compiler = ToolProvider.defaultJavaCompiler();
compiler.run(new FileInputStream("MyClass.java"),null, null);
```

Si noti inoltre che abbiamo catturato con il metodo `waitFor()` il codice di uscita del processo di compilazione. Se questo vale 0 allora il file è stato compilato correttamente. L'output del nostro programma sarà:

```
D:\esercizi>dir *.class

D:\esercizi>javac CompilatoreInterattivo.java
```

```
D:\esercizi>dir *.class
Il volume nell'unità C è OS
 Numero di serie del volume: 48D1-FF51

 Directory di D:\esercizi>

08/06/2014  15.11             1.693 CompilatoreInterattivo.class
               1 File          1.693 byte
               0 Directory  98.803.150.848 byte disponibili

D:\esercizi>java CompilatoreInterattivo
Digita il nome di un file java presente nella cartella corrente e batti
 enter, oppure scrivi "fine"
Moneta
Hai digitato MONETA!
Impossibile compilare Moneta
Moneta.java
Hai digitato MONETA.JAVA!
Moneta.java compilato!
fine
Fine programma!

D:\esercizi>dir *.class
 Il volume nell'unità C è OS
 Numero di serie del volume: 48D1-FF51

 Directory di D:\esercizi>

08/06/2014  13.47             1.693 CompilatoreInterattivo.class
08/06/2014  13.47             1.345 Moneta.class
08/06/2014  13.33               627 Valore$1.class
08/06/2014  13.33               636 Valore$2.class
08/06/2014  13.33               636 Valore$3.class
08/06/2014  13.33             1.786 Valore.class
               6 File          6.723 byte
               0 Directory  98.804.051.968 byte disponibili
```

Da riga di comando abbiamo prima cercato di visualizzare tutti i file .class con il comando dir *.class (cfr. appendice A), ma non ce n'erano. Poi abbiamo compilato la classe CompilatoreInterattivo.java e abbiamo controllato che il file CompilatoreInterattivo.class fosse stato generato. Poi abbiamo eseguito il nostro programma e nella nostra sessione interattiva abbiamo prima cercato di compilare il file Moneta, ma questo non esiste. Poi abbiamo provato a compilare il file Moneta.java e abbiamo verificato che effettivamente la compilazione sia andata a buon fine. Naturalmente è stata compilata anche la classe `Valore` e le sue classi anonime.

Soluzione 12.f)

Se parliamo di un array di interi, esistendo l'autoboxing-unboxing ci possiamo anche rifare ad un array di `Integer`. Visto che questa classe è dichiarata `final`, e che quindi non si può estendere, è impensabile crearne una sottoclasse che implementa `Comparable`. L'unica soluzione che ci rimane è creare una classe `Comparator` che ordina gli interi in maniera inversa:

```
import java.util.Comparator;

public class IntegerComparator implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return -(o1.compareTo(o2));
    }
}
```

La classe di test potrebbe limitarsi a questa:

```
import java.util.Arrays;

public class TestIntegerComparator {
    public static void main(String args[]) {
        Integer []array = {1942, 1947, 1971, 1984, 1976, 1974};
        Arrays.sort(array, new IntegerComparator());
        for (int intero : array) {
            System.out.println(intero);
        }
    }
}
```

Il cui output sarà:

```
1984
1976
1974
1971
1947
1942
```

Soluzione 12.g)

Il listato per la classe `WrapperComparable` dovrebbe essere definito così:

```
public class WrapperComparable implements Comparable<WrapperComparable> {

    private Integer integer;
```

```
    public WrapperComparable(Integer integer) {
        if (integer == null) {
            integer = 0;
        }
        this.integer = integer;
    }

    public Integer getInteger() {
        return integer;
    }

    public void setInteger(Integer integer) {
        this.integer = integer;
    }

    @Override
    public int compareTo(WrapperComparable otherWrapperComparable) {
        return -(integer.compareTo(otherWrapperComparable.getInteger()));
    }

    @Override
    public String toString() {
        return "WrapperComparable("+integer+ ")";
    }
}
```

Il listato per la classe `TestWrapperComparable` potrebbe essere il seguente:

```
import java.util.Arrays;

public class TestWrapperComparable {
    public static void main(String args[]) {
        WrapperComparable[] array = {new WrapperComparable(1942),
          new WrapperComparable(1974), new WrapperComparable(1907)};
        Arrays.sort(array);
        for (WrapperComparable wrapperComparable : array) {
            System.out.println(wrapperComparable);
        }
    }
}
```

Soluzione 12.h)

L'output della classe è:

```
false
true
true
true
false
```

Infatti `stringa1` e `stringa 2` sono due stringhe diverse, quindi è corretto che la prima istruzione di stampa, che confronta le stringhe con l'operatore `==` che si basa sugli indirizzi dei reference, stampi `false`. La seconda e la terza istruzione di stampa invece stamperanno `true`, perché il metodo `equals()` confronta i contenuti delle stringhe e non gli indirizzi dei reference. Il risultato della quarta e la quinta istruzioni di stampa invece, dipende dal fatto che la stringa `Claudio` è una stringa che è stata messa nel pool di stringhe (cfr. paragrafo 12.5.1) e il cui indirizzo coincide con l'indirizzo di `stringa1` (vedi la prima istruzione del metodo) ma non con quella di `stringa2`.

Soluzione 12.i)

Tutte le affermazioni sono corrette. In particolare nell'ultima si parla di modificare lo stato esterno di un oggetto, ma non esiste uno "stato esterno" da modificare.

Soluzione 12.l)

Tutte le affermazioni sono false! In particolare la numero 2 e la numero 3 sono scorrette perché un carattere di una stringa è immagazzinato in 16 bit (non una stringa), mentre se la stringa è compatta un suo carattere è immagazzinato utilizzando 8 bit. La numero 4 è scorretta perché l'opzione `-XX:-CompactStrings`, va utilizzata quando si esegue il programma non quando lo si compila.

Soluzione 12.m)

Il risultato giusto sarebbe `13`, ma la risposta corretta è l'ultima. Infatti nell'ultima istruzione mancano le parentesi tonde per invocare il metodo `length()`, il che provoca il seguente errore in compilazione:

```
Esercizio12M.java:10: error: cannot find symbol
        System.out.println(stringa.length);
                                   ^
  symbol:   variable length
  location: variable stringa of type String
1 error
```

Il compilatore ci avverte che non trova la variabile `length` (dato che mancano le parentesi tonde che caratterizzano la sintassi dei metodi). Si noti che tutte le istruzioni (tranne la penultima) non riassegnano il risultato del metodo invocato a `stringa`, che quindi punta sempre allo stesso oggetto immutabile dichiarato inizialmente. Solo nella penultima istruzione viene riassegnata alla variabile `stringa` un nuovo oggetto (che concatena un punto esclamativo alla fine della stringa) con

l'operatore `+=`. Ecco perché se non ci fosse l'errore all'ultimo statement, sarebbe stato stampato il valore `13`.

Soluzione 12.n)

L'output della classe `Esercizio12N` è il seguente:

```
intern()==equals()
```

quindi la risposta giusta è la numero 6. Infatti la chiamata al metodo `intern()` prova a recuperare l'oggetto `String` su cui è chiamato dal pool di stringhe, utilizzando il confronto che fornisce il metodo `equals()`. Nel caso nel pool non esista la stringa desiderata, questa viene aggiunta, e viene restituito un reference ad essa. Quindi la stringa viene aggiunta al pool, e le successive condizioni degli `if`, vengono verificate.

Soluzione 12.o)

L'output della classe `Esercizio12N` è il seguente:

```
123456

```

Si noti che l'output comprende l'andare a capo finale. Infatti dopo aver dato come valore `123789` alla variabile `stringa1`, otteniamo `stringa2` concatenando a `stringa1` un separatore di linea (che fa andare a capo) grazie al metodo statico `lineSeparator()` della classe `System`. Chiamando il metodo `toCharArray()` su `stringa2`, immagazziniamo in `array1` l'array di caratteri che costituivano la stringa `stringa2`. Quindi tale array ha dimensione 8 se eseguito su Windows (dove `System.lineSeparator()` contiene due caratteri `\r` e `\n`) mentre ha dimensione 7 su altre piattaforme come Linux (dove `System.lineSeparator()` contiene solo il carattere di escape `\n`). Poi all'array `array2` vengono assegnati tre elementi sempre di tipo carattere (`4`, `5` e `6`) che vengono copiati tutti tramite il metodo `System.arraycopy()` nell'`array1` a partire dall'indice `3`. Infine si stampa l'array `array1` e si esce dal programma tramite il metodo `System.exit()` (superfluo in questo caso perché il programma sarebbe terminato lo stesso subito dopo).

Soluzione 12.p)

La soluzione potrebbe essere implementata nella seguente maniera. Creiamo la seguente enumerazione che definisce semplicemente qualche tipologia di font.

```
public enum Font {
    ARIAL, TIMES_NEW_ROMAN, COURIER, MONOSPACED;
}
```

Poi creiamo la classe `TestoRTF` richiesta in questa maniera:

```
public class TestoRTF implements Cloneable {
    String testo;
    Font carattere;
    boolean sottolineato;

    public TestoRTF (String testo, Font carattere, boolean sottolineato) {
        this.testo = testo;
        this.carattere = carattere;
        this.sottolineato = sottolineato;
    }

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public String toString(){
        return "Testo = " + testo + ", carattere = " + carattere
          + ", sottolineato = " + sottolineato;
    }
}
```

Infine possiamo scrivere la classe di test in questa maniera:

```
public class Esercizio12P {
    public static void main(String args[]) throws CloneNotSupportedException {
        TestoRTF testoRTF1 = new TestoRTF("Java", Font.ARIAL, false);
        System.out.println(testoRTF1);
        System.out.println(testoRTF1.clone());
    }
}
```

Che produrrà il seguente output.

```
Testo = Java, carattere = ARIAL, sottolineato = false
Testo = Java, carattere = ARIAL, sottolineato = false
```

Soluzione 12.q)

Una delle possibili soluzioni è la ridefinizione della classe `Autenticazione`:

```
package com.claudiodesio.autenticazione;

import java.util.Scanner;

public class Autenticazione {
```

```
    public void login() {
        String username = System.getProperty("username");
        System.out.println(username);
        String password = System.getProperty("password");
        System.out.println(password);
        Utente utente = trovaUtente(username);
        if (utente != null) {
            if (verificaPassword(utente, password)) {
                Stampa.auguraBenvenuto(utente.getNome());
            } else {
                Stampa.autenticazioneFallita();
            }
        } else {
            Stampa.usernameInesistente();
        }
    }

    private Utente trovaUtente(String username) {
        Utente[] utenti = ProfiliUtenti.getInstance().getUtenti();
        if (username != null) {
            for (Utente utente : utenti) {
                if (username.equals(utente.getUsername())) {
                    return utente;
                }
            }
        }
        return null;
    }

    private boolean verificaPassword(Utente utente, String password) {
        boolean trovato = false;
        if (password != null) {
            if (password.equals(utente.getPassword())) {
                trovato = true;
            }
        }
        return trovato;
    }

    public static void main(String args[]) {
        Autenticazione autenticazione = new Autenticazione();
        autenticazione.login();
    }
}
```

Che eseguita con questi parametri:

```
java -Dusername=dansap -Dpassword=musica
  com.claudiodesio.autenticazione.Autenticazione
```

darà luogo al seguente output:

```
dansap
musica
Benvenuto Daniele
```

Soluzione 12.r)

Sono corrette le affermazioni numero 3 e 6. La numero 1 è falsa perché è possibile utilizzare anche altri algoritmi come SerialGC. Nell'affermazione numero 2 la prima parte della frase non è corretta: gli interventi della ParallelGC sono meno frequenti (ma più intensi). La "finalizzazione" consiste nel testare se esistono oggetti non più *raggiungibili* da qualche reference e quindi non utilizzabili, quindi anche l'affermazione 4 è errata. Il metodo `runFinalization()` richiede la finalizzazione degli oggetti, ma risiede nella classe `Runtime` non in `Object`, ecco perché anche l'affermazione 5 è errata.

Soluzione 12.s)

Una possibile soluzione è rappresentata dalla seguente classe:

```
import java.lang.reflect.*;

public class Esercizio12S {
    public static void main(String args[]) throws Exception {
        Class<ClasseConMembriPrivati> classe =
          ClasseConMembriPrivati.class;
        ClasseConMembriPrivati oggetto =
          classe.getDeclaredConstructor().newInstance();
        Field variabilePrivata = classe.getField("variabilePrivata");
        variabilePrivata.setAccessible(true);
        variabilePrivata.set(oggetto, "Variabile privata hackerata!");
        System.out.println(variabilePrivata.get(oggetto));
        Method metodoPrivata = classe.getMethod("metodoPrivato");
        metodoPrivata.setAccessible(true);
        metodoPrivata.invoke(oggetto);
    }
}
```

Il codice è molto intuitivo, tranne per il fatto che, nell'impostazione della variabile e nell'invocazione del metodo, è necessario passare anche l'oggetto su cui si vuole agire. L'output di questo programma si interrompe alla settima riga:

```
Exception in thread "main" java.lang.NoSuchFieldException:
  variabilePrivata
      at java.base/java.lang.Class.getField(Class.java:1956)
      at Esercizio12S.main(Esercizio12S.java:7)
```

Mentre se commentiamo le righe che fanno scattare l'eccezione e rieseguiamo il file (ovviamente dopo averlo ricompilato):

```
import java.lang.reflect.*;

public class Esercizio12S {
    public static void main(String args[]) throws Exception {
        Class<ClasseConMembriPrivati> classe =
          ClasseConMembriPrivati.class;
        ClasseConMembriPrivati oggetto =
          classe.getDeclaredConstructor().newInstance();
        /* Field variabilePrivata = classe.getField("variabilePrivata");
        variabilePrivata.setAccessible(true);
        variabilePrivata.set(oggetto, "Variabile privata hackerata!");
        System.out.println(variabilePrivata.get(oggetto));*/
        Method metodoPrivata = classe.getMethod("metodoPrivato");
        metodoPrivata.setAccessible(true);
        metodoPrivata.invoke(oggetto);
    }
}
```

otterremo il seguente output:

```
Exception in thread "main" java.lang.NoSuchMethodException:
  ClasseConMembriPrivati.metodoPrivato()
      at java.base/java.lang.Class.getMethod(Class.java:2065)
      at Esercizio12S.main(Esercizio12S.java:11)
```

Soluzione 12.t)

Basterà utilizzare la variabile `PI` (che rappresenta il π della classe `Math`) e il metodo `pow()` (che rappresenta la funzione potenza nel seguente modo):

```
public class Esercizio12T {
    public static void main(String args[]) {
        int raggio = 7;
        double area = Math.PI * Math.pow(raggio, 2);
        System.out.println("L'area delle circonferenza di raggio 7 è "
          + area);
    }
}
```

E questo è il suo output:

```
L'area delle circonferenza di raggio 7 è 153.93804002589985
```

Soluzione 12.u)

La risposta corretta è la 6, infatti l'output è il seguente:

```
Exception in thread "main" java.lang.NullPointerException
      at Eserczio12U.main(Eserczio12U.java:5)
```

Questo perché il risultato della chiamata del metodo `floor()` non viene riassegnato a nessuna variabile, e quindi la variabile `e` rimane con il valore iniziale (`2.718281828459045`). Quindi il metodo `check()` ritorna `null`, perché `2.0` è diverso da `2.718281828459045`. Ma `null` non si può assegnare ad un tipo primitivo `boolean`.

Soluzione 12.v)

L'output della classe Esercizio12.v, è il seguente:

```
long 128
```

Infatti il valore 128, come abbiamo asserito nel capitolo 2, viene considerato di default un `int`. Quindi, nonostante l'autoboxing sia sempre valido, viene invocato il metodo che ha come parametro un tipo primitivo, per ragioni di retro-compatibilità del codice, come spiegato nel paragrafo 12.6.1.4.

Soluzione 12.z)

La nostra soluzione consiste nel creare un'enumerazione `Segno`, che definisce i tre segni del gioco della morra cinese:

```
public enum Segno {
    SASSO, CARTA, FORBICI;
}
```

Poi abbiamo creato una classe che abbiamo chiamato `Regole`, che rappresenta il "nucleo di business" del gioco. Qui è presente l'algoritmo che definisce il vincitore. Abbiamo definito tale algoritmo implementando il metodo `compare()` dell'interfaccia `Comparator`. La scelta è ricaduta su tale metodo più per scopi didattici che per reale utilità, probabilmente esistevano soluzioni migliori:

```
import java.util.Comparator;

public class Regole implements Comparator<Segno> {
    @Override
    public int compare(Segno segno1, Segno segno2) {
        int risultato = 0;
        switch (segno1) {
            case CARTA: {
                if (segno2 == Segno.SASSO) {
                    risultato = 1;
```

```
                } else if (segno2 == Segno.FORBICI) {
                    risultato = -1;
                }
            }
            break;
            case SASSO: {
                if (segno2 == Segno.FORBICI) {
                    risultato = 1;
                } else if (segno2 == Segno.CARTA) {
                    risultato = -1;
                }
            }
            break;
            case FORBICI: {
                if (segno2 == Segno.CARTA) {
                    risultato = 1;
                } else if (segno2 == Segno.SASSO) {
                    risultato = -1;
                }
            }
            break;
            default: {
                risultato = 0;
            }
            break;

        }
        return risultato;
    }
}
```

La classe più importante è la classe `MorraCinese` che definisce un unico metodo pubblico `gioca()`, e tre metodi privati:

- `getSegno()` che recupera un elemento dell'enumerazione `Segno` in base alla sua posizione (`id`).
- `getSegnoComputer()` che riutilizza il metodo `getSegno()` passandogli un numero casuale tra `0` e `2`.
- `stabilisciVincitore()` che restituisce la stringa da stampare come output del programma, basandosi sulla comparazione definita nell'oggetto `Regole`.

```
import java.util.Random;

public class MorraCinese {

    public void gioca(int id) {
        Segno segnoGiocatore = getSegno(id);
```

```
        Segno segnoComputer = getSegnoComputer();
        System.out.println(segnoGiocatore + " VS " + segnoComputer);
        String risultato = stabilisciVincitore(segnoGiocatore, segnoComputer);
        System.out.println(risultato);
    }

    private String stabilisciVincitore(Segno segnoGiocatore,
                                       Segno segnoComputer) {
        Regole regole = new Regole();
        int risultato = regole.compare(segnoGiocatore, segnoComputer);
        if (risultato > 0) {
            return "Vince " + segnoGiocatore + "!\nHai vinto!";
        } else if (risultato < 0) {
            return "Vince " + segnoComputer + "!\nHai perso!";
        } else {
            return "Pari!";
        }
    }

    private Segno getSegno(int id) {
        Segno[] segni = Segno.values();
        Segno segnoComputer = segni[id];
        return segnoComputer;
    }

    private Segno getSegnoComputer() {
        Random random = new Random();
        return getSegno(random.nextInt(3));
    }
}
```

Infine la classe del `main()` gestisce un'eventuale input da riga di comando dell'utente e chiama il metodo `gioca()` di `MorraCinese`:

```
import java.util.Random;

public class Esercizio12Z {

    public static void main(String args[]) {
        int id = getId(args);
        MorraCinese morraCinese = new MorraCinese();
        morraCinese.gioca(id);
    }

    public static int getId(String args[]) {
        int id = 0;
        if (args.length != 0) {
            try {
                id = Integer.parseInt(args[0]);
                if (id < 0 || id > 2) {
                    System.out.println("Inserire un numero compreso "
                      + tra 0 e 2");
```

```
                    System.exit(1);
                }
            } catch (Exception exc) {
                System.out.println("Input non valido: " + args[0]);
                System.exit(1);
            }
        } else {
            id = new Random().nextInt(3);
        }
        return id;
    }
}
```

Ecco qualche esempio di output:

```
SASSO VS FORBICI
Vince SASSO!
Hai vinto!

FORBICI VS CARTA
Vince FORBICI!
Hai vinto!

SASSO VS SASSO
Pari!

FORBICI VS SASSO
Vince SASSO!
Hai perso!
```

Esercizi del capitolo 13

Tipi annotazioni

Con questi esercizi cercheremo di capire cosa significa sfruttare le annotazioni. Creeremo un'annotazione da zero, la sfrutteremo con un'applicazione creata ad hoc. Creeremo un verificatore di codice, che in base a delle annotazioni deciderà se le nostre classi rispettano determinati requisiti. Poi seguiranno anche tanti altri esercizi che supportano la preparazione alla certificazione, con quiz a risposta multipla.

Sino ad ora, per semplificare il nostro lavoro abbiamo fatto poco uso dei package, che però solitamente si usano sempre. Con le annotazioni useremo sempre i package per i nostri esercizi. Questo è necessario perché le annotazioni non sono rilevabili tramite reflection se non si trovano in un package. Quindi si consiglia di utilizzare un IDE come Eclipse o Netbeans.

Esercizio 13.a) Annotazioni, dichiarazioni ed uso, Vero o Falso:

1. Un'annotazione è un modificatore.
2. Un'annotazione è un'interfaccia.
3. Gli elementi di un'annotazione sembrano metodi astratti ma sottintendono un'implementazione implicita.

4. La seguente è una dichiarazione di annotazione valida:

```
public @interface MiaAnnotazione {
    void metodo();
}
```

5. La seguente è una dichiarazione di annotazione valida:

```
public @interface MiaAnnotazione {
    int metodo(int valore) default 5;
}
```

6. La seguente è una dichiarazione di annotazione valida:

```
public @interface MiaAnnotazione {
    int metodo() default -99;
    enum MiaEnum{VERO, FALSO};
    MiaEnum miaEnum();
}
```

7. Supponiamo che l'annotazione `MiaAnnotazione` definita nel punto 6 sia corretta. Con il seguente codice essa viene utilizzata correttamente:

```
public @MiaAnnotazione (
    MiaAnnotazione.MiaEnum.VERO
)
MiaAnnotazione.MiaEnum m() {
    return MiaAnnotazione.MiaEnum.VERO;
}
```

8. Supponiamo che l'annotazione `MiaAnnotazione` definita nel punto 6 sia corretta. Con il seguente codice essa viene utilizzata correttamente:

```
public @MiaAnnotazione (
    miaEnum=MiaAnnotazione.MiaEnum.VERO
)
MiaAnnotazione.MiaEnum m() {
    return @MiaAnnotazione.miaEnum;
}
```

9. Consideriamo la seguente annotazione.

```
public @interface MiaAnnotazione {
    int valore();
}
```

Con il seguente codice essa viene utilizzata correttamente:

```
public @MiaAnnotazione (
    5
)
void  m()
    //...
}
```

10. Consideriamo la seguente annotazione:

```
public @interface MiaAnnotazione {}
```

Con il seguente codice essa, viene utilizzata correttamente:

```
public @MiaAnnotazione void  m() {
    //...
}
```

Esercizio 13.b) Annotazioni e libreria, Vero o Falso:

1. La seguente annotazione è anche una meta-annotazione:

```
public @interface MiaAnnotazione ()
```

2. La seguente annotazione è anche una meta-annotazione:

```
@Target (ElementType.SOURCE)
public @interface MiaAnnotazione ()
```

3. La seguente annotazione è anche una meta-annotazione:

```
@Target (ElementType.@INTERFACE)
public @interface MiaAnnotazione ()
```

4. La seguente annotazione, se applicata ad un metodo, sarà documentata nella relativa documentazione Javadoc:

```
@Documented
@Target (ElementType.ANNOTATION_TYPE)
public @interface MiaAnnotazione ()
```

5. La seguente annotazione sarà ereditata se e solo se applicata ad una classe:

```
@Inherited
@Target (ElementType.METHOD)
public @interface MiaAnnotazione ()
```

6. Per la seguente annotazione è anche possibile creare un processore di annotazioni che riconosca al runtime il tipo di annotazione, per implementare un particolare comportamento:

```
@Documented
@Target (ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface MiaAnnotazione ()
```

7. `Override` è un'annotazione standard per segnalare al runtime di Java che un metodo effettua l'override di un altro.

8. `Deprecated` può essere considerata anche una meta-annotazione perché applicabile ad altre annotazioni.
9. `SuppressWarnings` è una annotazione a valore singolo. `Deprecated` e `Override` invece sono entrambe annotazioni marcatrici.
10. Non è possibile utilizzare contemporaneamente le `SuppressWarnings`, `Deprecated` e `Override` su di un'unica classe.

Esercizio 13.c)

Creare un'annotazione marcatrice per classi, di nome `Breve`, che deve essere usata per marcare classi che non hanno più di tre metodi. Questa annotazione deve appartenere ad un package `metadati`. Poi creare due classi `ClasseBreve` e `ClasseLunga`, la prima con un solo metodo, e l'altra con quattro metodi. Annotiamo entrambe le classi con `Breve`, e facciamole appartenere ad un package `dati`. Creiamo anche un'eccezione che chiameremo `AnnotationException` da far scattare quando una classe non rispetta la specifica dell'annotazione, che deve appartenere al package `eccezioni`.

Esercizio 13.d)

Creare una classe `VerificatoreInterattivo` che contiene un metodo `main()` che verifica se una classe specificata al runtime tramite l'utilizzo di una classe `Scanner` (già incontrata in precedenti esercizi) e annotata con `Breve`, soddisfa il requisito che abbiamo specificato nell'esercizio 13.c. Ovvero deve essere possibile specificare una classe, battere il tasto "Invio" e il programma deve stampare se la verifica è andata a buon fine o un eventuale messaggio di errore. Questa classe deve appartenere ad un package chiamato `test`.

Esercizio 13.e)

Creare un'annotazione a valore unico di nome `Specifica`, che deve essere usata per marcare le classi che hanno un numero preciso di variabili d'istanza. Inserire anche questa annotazione nel package `metadati`. Usare anche questa annotazione per le classi `ClasseBreve` e `ClasseLunga`, che per l'occasione dichiareranno delle variabili d'istanza. In particolare `ClasseBreve` dichiarerà una variabile d'istanza incapsulata, mentre `ClasseLunga` dichiarerà due variabili d'istanza non incapsulate.

Esercizio 13.f)

Modificare la classe `VerificatoreInterattivo` affinché verifichi la correttezza dell'utilizzo dell'annotazione `Specifica` come fatto per l'annotazione `Breve`.

Esercizio 13.g)

Creare un'annotazione completa di nome `Bean`, che deve essere usata per marcare le classi che hanno un costruttore senza parametri, le variabili incapsulate, un numero di metodi non superiore a un numero da specificare, e un numero di variabili non inferiore a un numero da specificare. Inserire anche questa annotazione nel package `metadati`. Usare anche questa annotazione per le classi `ClasseBreve` e `ClasseLunga`.

Esercizio 13.h)

Modificare la classe `VerificatoreInterattivo` affinché verifichi la correttezza dell'utilizzo dell'annotazione `Bean` come fatto per l'annotazione `Breve` e l'annotazione `Specifica`.

Esercizio 13.i)

Quali tra le seguenti affermazioni sono corrette?

1. L'annotazione `Override` è utile solo nella fase di compilazione.
2. Un'annotazione `Override` che annota un metodo precede sempre gli eventuali modificatori del metodo.
3. L'annotazione `Override` appartiene al package `java.lang`.
4. L'annotazione `Override` può annotare solo metodi.
5. L'annotazione `Override` è una annotazione marcatrice (marker).

Esercizio 13.l)

Considerata la seguente gerarchia:

```
public interface Giocatore {
    default void gioca() {}
}

public class Bambino implements Giocatore {
    /*INSERISCI CODICE QUI*/
}
```

Cosa è possibile inserire al posto del commento `/*INSERISCI CODICE QUI*/` tra i seguenti statement?

1. `@Override void gioca() {}`
2. `@Override public void gioca() {}`
3. `@Override public boolean equals(Object o) {return false;}`
4. `@Override public int hashcode() {return 1;}`
5. `@Override public String toString() {return "";}`

Esercizio 13.m)

Quali tra le seguenti affermazioni sono corrette?

1. L'annotazione `Deprecated` va utilizzata al posto del tag javadoc `@Deprecated`.
2. Un'annotazione `Deprecated` è di tipo annotazione completa, e dichiara due elementi: `since`, e `forRemoval`.
3. L'annotazione `Deprecated` appartiene al package `java.lang.annotation`.
4. L'annotazione `Deprecated` è una meta-annotazione.

Esercizio 13.n)

Quali tra le seguenti affermazioni sono corrette?

1. L'annotazione `FunctionalInterface` è utile solo nella fase di compilazione.
2. Un'annotazione `FunctionalInterface` dovrebbe annotare solo interfacce che hanno un unico metodo (che viene chiamato metodo SAM).
3. L'uso dell'annotazione `FunctionalInterface` è obbligatoria se l'interfaccia annotata ha un unico metodo.
4. L'annotazione `FunctionalInterface` è un'annotazione marcatrice (marker).
5. L'uso dell'annotazione `FunctionalInterface` implica sicuramente l'uso dell'annotazione `@Override`.

Esercizio 13.o)

Se compiliamo la seguente classe:

```
import java.util.*;

public class Esercizio13O {

    List objects;

    public Esercizio13O () {
        objects = new ArrayList();
    }

    public void rimuovi(Object object) {
        Iterator iterator = objects.iterator();
        if (iterator.hasNext()) {
            Object item = iterator.next();
            if (object.toString().equals(item.toString())) {
                iterator.remove();
            }
        }
    }
}
```

otterremo il seguente output che ci segnala tre warning.

```
Esercizio13O.java:4: warning: [rawtypes] found raw type: List
    List objects;
    ^
  missing type arguments for generic class List<E>
  where E is a type-variable:
    E extends Object declared in interface List
Esercizio13O.java:7: warning: [rawtypes] found raw type: ArrayList
        objects = new ArrayList();
                      ^
  missing type arguments for generic class ArrayList<E>
  where E is a type-variable:
    E extends Object declared in class ArrayList
Esercizio13O.java:12: warning: [rawtypes] found raw type: Iterator
        Iterator iterator = objects.iterator();
        ^
  missing type arguments for generic class Iterator<E>
  where E is a type-variable:
    E extends Object declared in interface Iterator
3 warnings
```

È possibile aggiungere un'annotazione per ottenere:

1. Solo due warning

2. Solo un warning
3. Nessun warning

Scrivere le tre versioni della classe che realizzano quanto richiesto.

Esercizio 13.p)

Quali tra le seguenti affermazioni sono corrette?

1. L'annotazione `Native` può essere usata per interfacciare Java con altri linguaggi.
2. Il termine "linguaggio nativo" significa linguaggio Java.
3. L'annotazione `Native` appartiene al package `java.lang`.
4. L'annotazione `Native` può annotare solo costanti.
5. L'annotazione `Native` è una annotazione marcatrice (marker).

Esercizio 13.q)

Quali tra le seguenti affermazioni sono corrette?

1. L'annotazione `Target` è una meta-annotazione che annota sé stessa.
2. L'annotazione `Target` è di tipo annotazione ordinaria. Infatti può specificare vari parametri.
3. `ElementType` è un'interfaccia che definisce i vari elementi di programmazione Java a cui si possono applicare le annotazioni.
4. L'annotazione `Target` può annotare anche annotazioni destinate ad annotare variabili locali e usi di tipi (come per esempio quando si definisce un cast, o si invoca un costruttore).
5. L'annotazione `Target` può annotare anche annotazioni destinate ad annotare istruzioni di `import`.

Esercizio 13.r)

Quali tra le seguenti affermazioni sono corrette?

1. L'annotazione `Retention` è annotata a sua volta da `Target`.
2. L'annotazione `Target` è annotata a sua volta da `Retention`.

3. L'annotazione `Documented` è annotata a sua volta da `Target`.
4. L'annotazione `Target` è annotata a sua volta da `Documented`.
5. L'annotazione `Retention` è annotata a sua volta da `Documented`.
6. L'annotazione `Documented` è annotata a sua volta da `Retention`.

Esercizio 13.s)

Quali tra le seguenti affermazioni sono corrette?

1. Con l'annotazione `Retention` decidiamo se l'annotazione annotata deve o meno essere conservata all'interno della classe compilata.
2. `RetentionPolicy` è un'enumerazione che dichiara solo due elementi: `SOURCE` e `CLASS`.
3. L'annotazione `Retention` e l'enumerazione `RetentionPolicy` appartengono al package `java.lang.annotation`.
4. L'annotazione `Retention` può annotare solo annotazioni.
5. L'annotazione `Retention` è di tipo annotazione ordinaria.

Esercizio 13.t)

Quali tra le seguenti affermazioni sono corrette?

1. L'annotazione `Documented` permette di includere all'interno della documentazione javadoc l'annotazione `Documented` stessa.
2. Un'annotazione `Documented` è annotata da sé stessa.
3. L'annotazione `Documented` è ereditata di default nelle sottoclassi.
4. L'annotazione `Documented` può annotare solo annotazioni che annotano classi.
5. L'annotazione `Documented` è una annotazione marcatrice (marker).

Esercizio 13.u)

Considerato il seguente codice:

```
import java.lang.annotation.*;
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Inherited
@Documented
public @interface Annotation13U {

}

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Inherited
@Documented
public @interface DifferentAnnotation {

}

@Annotation13U
public interface Interface13U {

}

@DifferentAnnotation
public class Esercizio13U implements Interface13U {

}
```

Qual è l'output del seguente programma?

```
import java.lang.reflect.*;
import java.util.*;
import java.lang.annotation.*;
public class AnnotationsReflection {
    public static void main(String[] args) throws Exception {
        Annotation[] dcs=Esercizio13U.class.getAnnotations();
        for (Annotation dc : dcs) {
            System.out.println(dc);
        }
    }
}
```

Esercizio 13.v)

Quali tra le seguenti affermazioni sono corrette?

1. L'annotazione `Repeatable` può annotare solo annotazioni, allo scopo di rendere ripetibile il loro utilizzo.
2. Quando si usa un'annotazione annotata correttamente con `Repeatable`, la

JVM crea al volo un oggetto per noi.

3. L'annotazione `Repeatable` deve essere per forza a valore singolo e ritornare un array.
4. Per dichiarare un'annotazione `Repeatable` bisogna creare anche un'altra annotazione ausiliare.
5. Un'annotazione annotata come `Repeatable` può annotare più volte lo stesso elemento di programmazione.

Esercizio 13.z)

Dato il seguente codice:

```
//...
@Check(">=0")
@Check("<100")
private int a;
//...
```

Creare l'annotazione `Check`.

Soluzioni degli esercizi capitolo 13

Soluzione 13.a) Annotazioni, dichiarazioni ed uso, Vero o Falso:

1. **Falso**, è un tipo annotazione.
2. **Falso**, è un tipo annotazione.
3. **Vero**.
4. **Falso**, un elemento di un'annotazione non può avere come tipo `void`.
5. **Falso**, un metodo di un'annotazione non può avere parametri in input.
6. **Vero**.
7. **Falso**, infatti è legale sia il codice del metodo `m()`, sia dichiarare `public` prima dell'annotazione (ma ovviamente è un modificatore del metodo). Non è legale però passare in input all'annotazione il valore `MiaAnnotazione.MiaEnum.VERO` senza specificare una sintassi del tipo chiave = valore.
8. **Falso**, infatti la sintassi:

   ```
   return @MiaAnnotazione.miaEnum;
   ```

   non è valida. Non si può utilizzare un'annotazione come se fosse una classe con variabili statiche pubbliche.
9. **Falso**, infatti l'annotazione in questione non è a valore singolo, perché il suo unico elemento non si chiama `value()`.
10. **Vero**.

Soluzione 13.b) Annotazioni e libreria, Vero o Falso:

1. **Vero**, infatti se non si specifica con la meta-annotazione `Target` quali sono gli elementi a cui è applicabile l'annotazione in questione, l'annotazione sarà di default applicabile a qualsiasi elemento tranne tipi parametro e uso dei tipi.
2. **Falso**, il valore `ElementType.SOURCE` non esiste.
3. **Falso**, il valore `ElementType.@INTERFACE` non esiste.
4. **Falso**, non è neanche applicabile a metodi per via del valore di `Target`, che è `ElementType.ANNOTATION_TYPE`.
5. **Falso**, infatti non può essere applicata ad una classe se è annotata con `@Target` (`ElementType.METHOD`).
6. **Vero**.
7. **Falso**, al compilatore, non al runtime.
8. **Vero**.
9. **Vero**.
10. **Vero**, `Override` non è applicabile a classi.

Soluzione 13.c)

Il listato dell'annotazione `Breve` dovrebbe essere simile al seguente:

```
package metadati;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
public @interface Breve {
}
```

Si noti che per i nostri scopi abbiamo dovuto specificare il livello di `Retention` a

`RetentionPolicy.RUNTIME`.
La classe `ClasseBreve` potrebbe invece essere modificata nel seguente modo:

```
package dati;

import metadati.Breve;

@Breve
public class ClasseBreve {
    public void metodo1() {
        System.out.println("metodo1");
    }
}
```

La classe `ClasseLunga` invece, potrebbe essere simile alla seguente:

```
package dati;

import metadati.Breve;

@Breve
public class ClasseLunga {
    public void metodo1() {
        System.out.println("metodo1");
    }
    public void metodo2() {
        System.out.println("metodo2");
    }
    public void metodo3() {
        System.out.println("metodo3");
    }
    public void metodo4() {
        System.out.println("metodo4");
    }
}
```

Infine ecco l'eccezione richiesta:

```
package eccezioni;

public class AnnotationException extends Exception {
    public AnnotationException(String msg) {
        super(msg);
    }

    @Override
    public String toString() {
        return "AnnotationException{" + getMessage() + "}";
    }
}
```

Soluzione 13.d)

Il listato del verificatore richiesto potrebbe essere il seguente:

```
package test;

import eccezioni.AnnotationException;
import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
import java.util.Scanner;
import metadati.Breve;

public class VerificatoreInterattivo {

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        String stringa = "";
        System.out.println("Digita il nome di un file java presente nella "
          + " cartella corrente e batti enter, oppure scrivi \"fine\" "
          + "per terminare il programma");
        while (!(stringa = scanner.next()).equals("fine")) {
            System.out.println("Hai digitato "
              + stringa.toUpperCase() + "!");
            try {
                verificaClasse(stringa);
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
        System.out.println("Fine programma!");
    }

    private static void verificaClasse(String stringa) throws Exception {
        Class objectClass = Class.forName(stringa);
        try {
            System.out.println("Inizio verifica annotazione @Breve per "
              + stringa);
            Annotation breve = objectClass.getAnnotation(Breve.class);
            if (breve != null) {
                Method[] methods = objectClass.getDeclaredMethods();
                final int numeroMetodi = methods.length;
                if (numeroMetodi > 3) {
                    throw new AnnotationException("Ci sono "
                      + numeroMetodi + " metodi nella classe "
                      + stringa);
                }
                System.out.println("Classe " + stringa +
                  " corretta!\nlista metodi:");
                for (Method method : methods) {
                    System.out.println(method);
```

```
                }
            } else {
                System.out.println("Questa classe non è annotata con "
                  + "@Breve");
            }
        } finally {
            System.out.println("Fine verifica annotazione @Breve per "
              + stringa);
        }
    }
}
```

In pratica si tratta solo di concentrarsi sul metodo `verificaClasse()`, il cui codice è comunque molto intuitivo.

Il seguente output viene prodotto specificando prima le due classi davvero annotate con `@Breve`, poi una che non è annotata (la classe `AnnotationException`) e infine una classe inesistente:

```
Digita il nome di un file java presente nella  cartella corrente e
 batti enter, oppure scrivi "fine" per terminare il programma
dati.ClasseBreve
Hai digitato DATI.CLASSEBREVE!
Inizio verifica annotazione @Breve per dati.ClasseBreve
Classe dati.ClasseBreve corretta!
lista metodi:
public void dati.ClasseBreve.metodo1()
public void dati.ClasseBreve.setVariabile(java.lang.String)
public java.lang.String dati.ClasseBreve.getVariabile()
Fine verifica annotazione @Breve per dati.ClasseBreve
dati.ClasseLunga
Hai digitato DATI.CLASSELUNGA!
Inizio verifica annotazione @Breve per dati.ClasseLunga
Fine verifica annotazione @Breve per dati.ClasseLunga
AnnotationException{Ci sono 4 metodi nella classe dati.ClasseLunga}
      at
  test.VerificatoreInterattivo.verificaClasse(
  VerificatoreInterattivo.java:37)
      at
  test.VerificatoreInterattivo.main(VerificatoreInterattivo.java:20)

eccezioni.AnnotationException
Hai digitato ECCEZIONI.ANNOTATIONEXCEPTION!
Inizio verifica annotazione @Breve per eccezioni.AnnotationException
Questa classe non è annotata con @Breve
Fine verifica annotazione @Breve per eccezioni.AnnotationException

ClasseInesistente
Hai digitato CLASSEINESISTENTE!
java.lang.ClassNotFoundException: ClasseInesistente
```

```
    at java.net.URLClassLoader$1.run(URLClassLoader.java:372)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:361)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:360)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    at java.lang.Class.forName0(Native Method)
    at java.lang.Class.forName(Class.java:259)
    at test.VerificatoreInterattivo.verificaClasse(
VerificatoreInterattivo.java:29)
    at
test.VerificatoreInterattivo.main(VerificatoreInterattivo.java:20)
```

Soluzione 13.e)

Il listato dell'annotazione `Specifica` dovrebbe essere il seguente:

```
package metadati;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
public @interface Specifica {
    int value();
}
```

Il listato delle classi `ClasseBreve` e `ClasseLunga` potrebbe essere modificato come segue:

```
package dati;

import metadati.Breve;
import metadati.Specifica;

@Breve
@Specifica(1)
public class ClasseBreve {

    public void metodo1() {
        System.out.println("metodo1");
    }
```

```
    private String variabile;

    public String getVariabile() {
        return variabile;
    }

    public void setVariabile(String variabile) {
        this.variabile = variabile;
    }
}
```

e:

```
package dati;

import metadati.Specifica;

@Breve
@Specifica(3)
public class ClasseLunga {

    public String variabile1;
    public String variabile2;

    public void metodo1() {
        System.out.println("metodo1");
    }

    public void metodo2() {
        System.out.println("metodo2");
    }

    public void metodo3() {
        System.out.println("metodo3");
    }

    public void metodo4() {
        System.out.println("metodo4");
    }
}
```

Soluzione 13.f)

Modifichiamo il codice del nostro verificatore, estraendo come metodo il codice di verifica dell'annotazione `Breve`, e commentandone la chiamata. Creiamo un equivalente metodo per verificare l'annotazione `Specifica`. Di seguito riportiamo solo il codice modificato:

```
    private static void verificaClasse(String stringa) throws Exception {
        Class objectClass = Class.forName(stringa);
```

```
        //verificaBreve(stringa, objectClass);
        verificaSpecifica(stringa, objectClass);
    }

    private static void verificaBreve(String stringa, Class objectClass) throws
      AnnotationException {
        try {
            System.out.println("Inizio verifica annotazione @Breve per "
              + stringa);
            Annotation breve = objectClass.getAnnotation(Breve.class);
            if (breve != null) {
                Method[] methods = objectClass.getDeclaredMethods();
                final int numeroMetodi = methods.length;
                if (numeroMetodi > 3) {
                    throw new AnnotationException("Ci sono "
                      + numeroMetodi + " metodi nella classe "
                      + stringa);
                }
                System.out.println("Classe " + stringa +
                  " corretta!\nlista metodi:");
                for (Method method : methods) {
                    System.out.println(method);
                }
            } else {
                System.out.println("Questa classe non è annotata con "
                  + "@Breve");
            }
        } finally {
            System.out.println("Fine verifica annotazione @Breve per "
              + stringa);
        }
    }

    private static void verificaSpecifica(String stringa, Class objectClass)
      throws AnnotationException {
        try {
            System.out.println("Inizio verifica annotazione @Specifica per "
              + stringa);
            Specifica specifica = (Specifica)objectClass.getAnnotation(
              Specifica.class);
            if (specifica != null) {
                int numeroVariabiliDaSpecifica = specifica.value();
                Field[] fields = objectClass.getDeclaredFields();
                final int numeroVariabili = fields.length;
                if (numeroVariabili != numeroVariabiliDaSpecifica) {
                    throw new AnnotationException("Ci sono "
                      + numeroVariabili
                      + " variabili nella classe " + stringa +
                      " ma dovrebbero essere " +
                      numeroVariabiliDaSpecifica);
                }
```

```
                    System.out.println("Classe " + stringa +
                      " corretta!\nlista variabili:");
                    for (Field field : fields) {
                        System.out.println(field);
                    }
                } else {
                    System.out.println(
                      "Questa classe non è annotata con @Specifica");
                }
            } finally {
                System.out.println("Fine verifica annotazione @Breve per "
                  + stringa);
            }
        }
```

Ecco un esempio di output:

```
Digita il nome di un file java presente nella cartella corrente e
  batti enter, oppure scrivi "fine" per terminare il programma
dati.ClasseBreve
Hai digitato DATI.CLASSEBREVE!
Inizio verifica annotazione @Specifica per dati.ClasseBreve
Classe dati.ClasseBreve corretta!
lista variabili:
private java.lang.String dati.ClasseBreve.variabile
Fine verifica annotazione @Breve per dati.ClasseBreve
dati.ClasseLunga
Hai digitato DATI.CLASSELUNGA!
Inizio verifica annotazione @Specifica per dati.ClasseLunga
Fine verifica annotazione @Breve per dati.ClasseLunga
AnnotationException{Ci sono 2 variabili nella classe dati.ClasseLunga
  ma dovrebbero essere 3}
      at
  test.VerificatoreInterattivo.verificaSpecifica(
  VerificatoreInterattivo.java:69)
      at
  test.VerificatoreInterattivo.verificaClasse(
  VerificatoreInterattivo.java:33)
      at
  test.VerificatoreInterattivo.main(VerificatoreInterattivo.java:22)
```

Soluzione 13.g)

Il listato dell'annotazione `Bean` potrebbe essere il seguente:

```
  package metadati;

  import java.lang.annotation.Documented;
  import java.lang.annotation.ElementType;
  import java.lang.annotation.Inherited;
```

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
public @interface Bean {
    int numeroMassimoMetodi();
    int numeroMinimoVariabili();
}
```

Il listato delle classi `ClasseBreve` e `ClasseLunga` potrebbe essere modificato come segue:

```
package dati;

import metadati.Bean;
import metadati.Breve;
import metadati.Specifica;

@Breve
@Specifica(1)
@Bean(numeroMassimoMetodi = 10, numeroMinimoVariabili = 1)
public class ClasseBreve {

    public void metodo1() {
        System.out.println("metodo1");
    }

    private String variabile;

    public String getVariabile() {
        return variabile;
    }

    public void setVariabile(String variabile) {
        this.variabile = variabile;
    }
}
```

e:

```
package dati;

import metadati.Bean;
import metadati.Breve;
import metadati.Specifica;

@Breve
```

```
@Specifica(3)
@Bean(numeroMassimoMetodi = 10, numeroMinimoVariabili = 1)
public class ClasseLunga {

    public String variabile1;
    public String variabile2;

    public void metodo1() {
        System.out.println("metodo1");
    }

    public void metodo2() {
        System.out.println("metodo2");
    }

    public void metodo3() {
        System.out.println("metodo3");
    }

    public void metodo4() {
        System.out.println("metodo4");
    }
}
```

Soluzione 13.h)

Modifichiamo il codice del nostro verificatore e, sul modello della soluzione dell'esercizio 13.f, creiamo un metodo per verificare l'annotazione `Bean`. Di seguito riportiamo solo il codice modificato:

```
    private static void verificaClasse(String stringa) throws Exception {
        Class objectClass = Class.forName(stringa);
        //verificaBreve(stringa, objectClass);
        //verificaSpecifica(stringa, objectClass);
        verificaBean(stringa, objectClass);
    }

    private static void verificaBreve(String stringa, Class objectClass)
      throws AnnotationException {
        try {
            System.out.println("Inizio verifica annotazione @Breve per "
              + stringa);
            Annotation breve = objectClass.getAnnotation(Breve.class);
            if (breve != null) {
                Method[] methods = objectClass.getDeclaredMethods();
                final int numeroMetodi = methods.length;
                if (numeroMetodi > 3) {
                    throw new AnnotationException("Ci sono "
                      + numeroMetodi + " metodi nella classe "
```

```
                    + stringa);
            }
            System.out.println("Classe " + stringa +
              " corretta!\nlista metodi:");
            for (Method method : methods) {
                System.out.println(method);
            }
        } else {
            System.out.println("Questa classe non è annotata con "
              + "@Breve");
        }
    } finally {
        System.out.println("Fine verifica annotazione @Breve per "
          + stringa);
    }
}

private static void verificaSpecifica(String stringa, Class objectClass)
  throws AnnotationException {
    try {
        System.out.println("Inizio verifica annotazione @Specifica per "
          + stringa);
        Specifica specifica = (Specifica)
                objectClass.getAnnotation(Specifica.class);
        if (specifica != null) {
            int numeroVariabiliDaSpecifica = specifica.value();
            Field[] fields = objectClass.getDeclaredFields();
            final int numeroVariabili = fields.length;
            if (numeroVariabili != numeroVariabiliDaSpecifica) {
                throw new AnnotationException("Ci sono "
                  + numeroVariabili
                  + " variabili nella classe " + stringa
                  + " ma dovrebbero essere "
                  + numeroVariabiliDaSpecifica);
            }
            System.out.println("Classe " + stringa
              + " corretta!\nlista variabili:");
            for (Field field : fields) {
                System.out.println(field);
            }
        } else {
            System.out.println(
              "Questa classe non è annotata con @Specifica");
        }
    } finally {
        System.out.println("Fine verifica annotazione @Breve per "
          + stringa);
    }
}

private static void verificaBean(String stringa, Class objectClass)
```

```
        throws AnnotationException, NoSuchMethodException {
          try {
              System.out.println("Inizio verifica annotazione @Bean per "
                + stringa);
              Bean bean = (Bean) objectClass.getAnnotation(Bean.class);
              if (bean != null) {
                  controlloNumeroVariabili(bean, objectClass, stringa);
                  controlloNumeroMetodi(bean, objectClass, stringa);
                  controlloCostruttoreSenzaParametro(objectClass, stringa);
                  controlloIncapsulamento(objectClass, stringa);
              } else {
                  System.out.println(
                    "Questa classe non è annotata con @Specifica");
              }
          } finally {
              System.out.println(
                "Fine verifica annotazione @Breve per " + stringa);
          }
      }

      private static void controlloNumeroVariabili(Bean bean, Class objectClass,
        String stringa) throws AnnotationException {
          int numeroMinimoVariabili = bean.numeroMinimoVariabili();
          Field[] fields = objectClass.getDeclaredFields();
          final int numeroVariabili = fields.length;
          if (numeroVariabili < numeroMinimoVariabili) {
              throw new AnnotationException("Ci sono " + numeroVariabili
                + " variabili nella classe " + stringa
                + " ma dovrebbero essere almeno "
                + numeroMinimoVariabili);
          }
          System.out.println("Classe " + stringa
            + " numero variabili ok!\nlista variabili:");
          for (Field field : fields) {
              System.out.println(field);
          }
      }

      private static void controlloNumeroMetodi(Bean bean, Class objectClass,
        String stringa) throws AnnotationException, NoSuchMethodException {
          int numeroMassimoMetodi = bean.numeroMassimoMetodi();
          Method[] methods = objectClass.getDeclaredMethods();
          final int numeroMetodi = methods.length;
          if (numeroMetodi > numeroMassimoMetodi) {
              throw new AnnotationException("Ci sono " + numeroMetodi
                + " metodi nella classe " + stringa
                + " ma dovrebbero essere al massimo "
                + numeroMassimoMetodi);
          }
          System.out.println("Classe " + stringa
            + " numero metodi ok!\nlista metodi:");
```

```
            for (Method method : methods) {
                System.out.println(method);
            }
        }

        private static void controlloCostruttoreSenzaParametro(Class objectClass,
          String stringa) throws AnnotationException, NoSuchMethodException {
            Constructor constructor = objectClass.getConstructor();
            if (constructor == null) {
                throw new AnnotationException(
                  "Niente costruttore senza parametri!");
            }
            System.out.println("Classe " + stringa
              + " costruttore senza parametri presente!:");
            System.out.println(constructor);
        }

        private static void controlloIncapsulamento(Class objectClass,
          String stringa) throws AnnotationException, NoSuchMethodException {
            Field[] fields = objectClass.getDeclaredFields();
            for (Field field : fields) {
                final String nomeVariabile = field.getName();
                final Class<?> type = field.getType();
                final Method setMethod = objectClass.getDeclaredMethod("set"
                  + capitalize(nomeVariabile), type);
                final Method getMethod = objectClass.getDeclaredMethod("get"
                  + capitalize(nomeVariabile));
                if (setMethod == null || getMethod == null ||
                    !getMethod.getReturnType().equals(type)) {
                    throw new AnnotationException("Variabile " + nomeVariabile
                      + " non incapsulata correttamente nella classe "
                      + stringa);

                }
            }
            System.out.println("Classe " + stringa + " incapsulamento ok!");
        }

        private static String capitalize(String string) {
            return string.substring(0, 1).toUpperCase() + string.substring(1);
        }
    }
```

Ecco un esempio di output:

```
Digita il nome di un file java presente nella cartella corrente e
  batti enter, oppure scrivi "fine" per terminare il programma
dati.ClasseBreve
Hai digitato DATI.CLASSEBREVE!
Inizio verifica annotazione @Bean per dati.ClasseBreve
Classe dati.ClasseBreve numero variabili ok!
```

```
lista variabili:
private java.lang.String dati.ClasseBreve.variabile
Classe dati.ClasseBreve numero todi ok!
lista metodi:
public void dati.ClasseBreve.metodo1()
public java.lang.String dati.ClasseBreve.getVariabile()
public void dati.ClasseBreve.setVariabile(java.lang.String)
Classe dati.ClasseBreve costruttore senza parametri presente!:
public dati.ClasseBreve()
Classe dati.ClasseBreve incapsulamento ok!
Fine verifica annotazione @Breve per dati.ClasseBreve
dati.ClasseLunga
Hai digitato DATI.CLASSELUNGA!
Inizio verifica annotazione @Bean per dati.ClasseLunga
Classe dati.ClasseLunga numero variabili ok!
lista variabili:
public java.lang.String dati.ClasseLunga.variabile1
public java.lang.String dati.ClasseLunga.variabile2
Classe dati.ClasseLunga numero todi ok!
lista metodi:
public void dati.ClasseLunga.metodo3()
public void dati.ClasseLunga.metodo2()
public void dati.ClasseLunga.metodo4()
java.lang.NoSuchMethodException:
  dati.ClasseLunga.setVariabile1(java.lang.String)
public void dati.ClasseLunga.metodo1()
      at java.lang.Class.getDeclaredMethod(Class.java:2117)
Classe dati.ClasseLunga costruttore senza parametri presente!:
public dati.ClasseLunga()
      at test.VerificatoreInterattivo.controlloIncapsulamento(
  VerificatoreInterattivo.java:152)
Fine verifica annotazione @Breve per dati.ClasseLunga
      at test.VerificatoreInterattivo.verificaBean(
  VerificatoreInterattivo.java:97)
      at test.VerificatoreInterattivo.verificaClasse(
  VerificatoreInterattivo.java:36)
      at test.VerificatoreInterattivo.main(
  VerificatoreInterattivo.java:24)
Digita il nome di un file java presente nella cartella corrente e
  batti enter, oppure scrivi "fine" per terminare il programma
dati.ClasseBreve
Hai digitato DATI.CLASSEBREVE!
Inizio verifica annotazione @Bean per dati.ClasseBreve
Classe dati.ClasseBreve numero variabili ok!
lista variabili:
private java.lang.String dati.ClasseBreve.variabile
Classe dati.ClasseBreve numero todi ok!
lista metodi:
public void dati.ClasseBreve.metodo1()
public java.lang.String dati.ClasseBreve.getVariabile()
public void dati.ClasseBreve.setVariabile(java.lang.String)
```

```
Classe dati.ClasseBreve costruttore senza parametri presente!:
public dati.ClasseBreve()
Classe dati.ClasseBreve incapsulamento ok!
Fine verifica annotazione @Breve per dati.ClasseBreve
dati.ClasseLunga
Hai digitato DATI.CLASSELUNGA!
Inizio verifica annotazione @Bean per dati.ClasseLunga
Classe dati.ClasseLunga numero variabili ok!
lista variabili:
public java.lang.String dati.ClasseLunga.variabile1
public java.lang.String dati.ClasseLunga.variabile2
Classe dati.ClasseLunga numero todi ok!
lista metodi:
public void dati.ClasseLunga.metodo3()
public void dati.ClasseLunga.metodo2()
public void dati.ClasseLunga.metodo4()
dati.ClasseLunga.setVariabile1(java.lang.String)
public void dati.ClasseLunga.metodo1()
Classe dati.ClasseLunga costruttore senza parametri presente!:
java.lang.NoSuchMethodException:
      at java.lang.Class.getDeclaredMethod(Class.java:2117)
public dati.ClasseLunga()
      at test.VerificatoreInterattivo.controlloIncapsulamento(
  VerificatoreInterattivo.java:152)
Fine verifica annotazione @Breve per dati.ClasseLunga
      at test.VerificatoreInterattivo.verificaBean(
  VerificatoreInterattivo.java:97)
      at test.VerificatoreInterattivo.verificaClasse(
  VerificatoreInterattivo.java:36)
      attest.VerificatoreInterattivo.main(VerificatoreInterattivo.java:24)
```

Soluzione 13.i)

Tutte le affermazioni sull'annotazione `Override` sono corrette.

Soluzione 13.l)

Solo lo statement 1 non è corretto in quanto, non esplicitando il modificatore `public`, si sta rendendo il metodo che applica l'override, meno accessibile rispetto all'originale ereditato. Quest'ultimo infatti, essendo un metodo dichiarato in un'interfaccia, è implicitamente pubblico. Insomma è stata violata la regola di cui abbiamo parlato nel paragrafo 8.2.3 (terza regola).
Infatti l'eventuale output sarebbe il seguente:

```
Bambino.java:2: error: gioca() in Bambino cannot implement gioca() in
  Giocatore
@Override void gioca(){}    /*INSERISCI CODICE QUI*/
              ^
```

```
  attempting to assign weaker access privileges; was public
1 error
```

Soluzione 13.m)

Solo la seconda affermazione è corretta. La prima non è corretta perché semmai l'annotazione e il tag javadoc vanno usati in contemporanea. La terza è anch'essa scorretta perché l'annotazione appartiene a `java.lang`. Infine la numero 4 è sbagliata perché tale annotazione può annotare diversi elementi della programmazione Java, ma non altre anntoazioni.

Soluzione 13.n)

Le affermazioni corrette sono la 1, la 2 e la 4. La terza è errata visto che è possibile dichiarare un un'interfaccia funzionale senza obbligatoriamente annotarla.
La numero 5 è errata invece, perché come `FunctionalInterface` anche l'annotazione `@Override` è sempre facoltativa.

Soluzione 13.o)

Per ottenere solo due warning vi sono più soluzioni: aggiungere un'annotazione per sopprimere le annotazioni o sul costruttore, o sulla variabile d'istanza `objects`, o sul metodo `rimuovi()`. Per esempio scegliamo la variabile d'istanza `objects`:

```
import java.util.*;

public class Esercizio13O {

    @SuppressWarnings({"rawtypes"})
    List objects;

    public Esercizio13O() {
        objects = new ArrayList();
    }

    @SuppressWarnings({"rawtypes"})
    public void rimuovi(Object object) {
        Iterator iterator = objects.iterator();
        if (iterator.hasNext()) {
            Object item = iterator.next();
            if (object.toString().equals(item.toString())) {
                iterator.remove();
            }
        }
    }
}
```

Che produce il seguente output:

```
Esercizio130.java:8: warning: [rawtypes] found raw type: ArrayList
        objects = new ArrayList();
                      ^
  missing type arguments for generic class ArrayList<E>
  where E is a type-variable:
    E extends Object declared in class ArrayList
Esercizio130.java:13: warning: [rawtypes] found raw type: Iterator
        Iterator iterator = objects.iterator();
        ^
  missing type arguments for generic class Iterator<E>
  where E is a type-variable:
    E extends Object declared in interface Iterator
2 warnings
```

È facile immaginare come ottenere un solo warning. Per esempio, annotiamo il costruttore e il metodo `rimuovi()`:

```
import java.util.*;

public class Esercizio130 {

    List objects;

    @SuppressWarnings({"rawtypes"})
    public Esercizio130() {
        objects = new ArrayList();
    }

    @SuppressWarnings({"rawtypes"})
    public void rimuovi(Object object) {
        Iterator iterator = objects.iterator();
        if (iterator.hasNext()) {
            Object item = iterator.next();
            if (object.toString().equals(item.toString())) {
                iterator.remove();
            }
        }
    }
}
```

In questo caso otterremo il seguente output:

```
Esercizio130.java:5: warning: [rawtypes] found raw type: List
    List objects;
    ^
  missing type arguments for generic class List<E>
  where E is a type-variable:
    E extends Object declared in interface List
1 warning
```

Infine è ovvio che annotando anche la variabile d'istanza non otterremo nessun warning, ma è più comodo annotare l'intera classe:

```
import java.util.*;

@SuppressWarnings({"rawtypes"})
public class Esercizio13O {

    List objects;

    public Esercizio13O() {
        objects = new ArrayList();
    }

    public void rimuovi(Object object) {
        Iterator iterator = objects.iterator();
        if (iterator.hasNext()) {
            Object item = iterator.next();
            if (object.toString().equals(item.toString())) {
                iterator.remove();
            }
        }
    }
}
```

Che non produce warning in compilazione.

Soluzione 13.p)

Le affermazioni corrette sono 1, 4 e 5. Con il termine "linguaggio nativo" ci si riferisce solitamente al linguaggio utilizzato dalla piattaforma su cui gira il programma, ovvero il sistema operativo, che solitamente coincide con C/C++, quindi l'affermazione numero 2 è errata.
L'annotazione `Native` appartiene al package `java.lang.annotation`, quindi anche l'affermazione numero 3 è scorretta.

Soluzione 13.q)

Le affermazioni corrette sono 1 e 4. La numero 2 non è corretta perché `Target` è un'annotazione a valore singolo di tipo array di `ElementType`.
La numero 3 non è corretta perché `ElementType` non è un'interfaccia ma un'enumerazione.
La numero 5 non è corretta perché esistono delle situazioni non contemplate dalle interfacce annotate con `Target` che specifica come `ELEMENT_TYPE TYPE_USE`, e tra queste c'è proprio il caso dell'`import` come abbiamo specificato nelle ultime righe del paragrafo 13.2.1.1.

Soluzione 13.r)

Tutte le affermazioni sono vere.

Soluzione 13.s)

Le affermazioni corrette sono 1, 3 e 4.
La numero 2 non è corretta perché `RetentionPolicy` dichiara anche un terzo elemento (`RUNTIME`).
La numero 5 non è corretta perché `Target` è un'annotazione a valore singolo di tipo `RetentionPolicy`.

Soluzione 13.t)

Le affermazioni corrette sono la 2 e la 5.
La numero 1 non è corretta perché l'annotazione `Documented` è un'annotazione che annota annotazioni riportate all'interno della documentazione javadoc generata.
Le numero 3 e 4 sono semplicemente inventate.

Soluzione 13.u)

Il programma che legge le annotazioni di `Esercizio13U` stamperà solo l'annotazione che è stata dichiarata per la classe stessa, ma non erediterà quella dall'interfaccia `Interface13U`. Infatti l'annotazione `Inherited` funziona solo sulle classi e non sulle interfacce.
Ecco l'output:

```
@DifferentAnnotation()
```

Soluzione 13.v)

Le affermazioni corrette sono 1, 2, 4 e 5.
La numero 3 non è corretta perché l'annotazione a valore singolo che deve ritornare un array, quella che abbiamo chiamato annotazione contenitore, è l'annotazione ausiliaria di cui si parla nell'affermazione 4.

Soluzione 13.z)

La soluzione potrebbe consistere nel creare le seguenti annotazioni: l'annotazione `Check`, e la sua annotazione contenitore `Checks`:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
@Inherited
@Documented
@Repeatable(Checks.class)
public @interface Check {

    String value();
}
```

e

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
@Inherited
@Documented
public @interface Checks {

    Check [] value();
}
```

Esercizi del capitolo 14

Le librerie di utilità: il package `java.util` e Date-Time API

Il capitolo è dedicato essenzialmente al package `java.util` e la nuova libreria "Date & Time API". Il package `java.util` è vastissimo, esistono tantissime altre classi ed interfacce degne di nota. Utilizzare la documentazione, anche per svolgere questi esercizi, è fondamentale.

Esercizio 14.a) Package `java.util`, Vero o Falso:

1. La classe `Properties` estende `Hashtable` ma consente di salvare su un file le coppie chiave-valore rendendole persistenti.
2. La classe `Locale` astrae il concetto di "zona".
3. La classe `ResourceBundle` rappresenta un file di properties che permette di gestire l'internazionalizzazione. Il rapporto tra nome del file e `Locale` specificato per individuare tale file permetterà di gestire la configurazione della lingua delle nostre applicazioni.
4. L'output del seguente codice:

```
StringTokenizer st = new StringTokenizer(
  "Il linguaggio object oriented Java", "t", false);
```

```
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

sarà:

```
Il linguaggio objec
t
orien
t
ed Java
```

5. Il seguente codice non è valido:

```
Pattern p = Pattern.compile("\bb");
Matcher m = p.matcher("blablabla...");
boolean b = m.find();
System.out.println(b);
```

6. La classe `Preferences` permette di gestire file di configurazione con un file XML.

7. La classe `Formatter` definisce il metodo `printf()`.

8. L'espressione regolare `[a]` è un quantificatore (greedy quantifier).

9. Il seguente codice:

```
Date d = new Date();
d.now();
```

crea un oggetto `Date` con la data attuale.

10. Un `SimpleNumberFormat` può formattare e analizzare qualsiasi valuta grazie alla specifica di un `Locale`.

Esercizio 14.b) Date-Time API, Vero o Falso:

1. I metodi di tipo "from" permettono di recuperare un certo tipo temporale, a partire da un tipo temporale con più informazioni.

2. L'unico modo per fare il parsing di una stringa per ottenere un oggetto `Instant`, è passare tramite la classe `DateTimeFormatter`.

3. La classe `Duration` calcola la distanza tra due istanti, quindi è definita sulla timeline.

4. Non è possibile memorizzare informazioni sull'orario in un oggetto di tipo `LocalDate`.

5. È possibile memorizzare informazioni sulla data in un oggetto di tipo `LocalTime`.
6. È possibile memorizzare informazioni sulla data in un oggetto di tipo `ZonedDate`.
7. Il metodo `between()` di `ChronoUnit` restituisce un oggetto `Duration`.
8. I temporal adjusters possono essere passati a metodi di tipo "with" per compiere operazioni su date e orari.
9. I temporal queries possono essere passati a metodi di tipo "with" per recuperare informazioni su date e orari.
10. In generale è possibile sostituire la classe `Date` con la classe `Instant`.

Esercizio 14.c)

Creare una classe `Traduttore` che espone un metodo `traduci()` per tradurre un numero limitato di parole dall'inglese all'italiano e viceversa, utilizzando un resource bundle.

Esercizio 14.d)

Creare una classe `TestTraduttore` per testare la correttezza delle traduzioni.

Esercizio 14.e)

Creare una classe `StringUtils` che dichiara solo metodi statici (e che quindi è inutile istanziare). Essa deve esporre un metodo `ricerca()`, che tramite espressioni regolari deve ricercare tutte le parole in un testo (specificato in input) che iniziano con un certo carattere, e restituirle all'interno di una lista.

Esercizio 14.f)

Creare una classe `TestStringUtils` per testare la correttezza del metodo `ricerca()`.

Esercizio 14.g)

Creare una classe `DateUtils` che dichiara solo metodi statici (e che quindi è inutile istanziare). Questa classe, dati due istanti in input, deve essere capace di restituire il numero di:

- secondi
- minuti
- ore
- giorni
- settimane
- mesi

che intercorrono tra i due istanti.

Esercizio 14.h)

Modificare la classe `DateUtils` per dotarla di un metodo che, dato un istante in input, deve essere capace di restituire il numero di:

- secondi
- minuti
- ore
- giorni
- settimane
- mesi

che intercorrono tra l'istante specificato e l'istante attuale.

Esercizio 14.i)

Nella classe `DateUtils` creare un metodo che restituisca l'ora esatta nel formato "HH:mm ss".

Esercizio 14.l)

Nella classe `DateUtils` creare un metodo che formatti la data specificata secondo il pattern specificato in input.

Esercizio 14.m)

Nella classe `DateUtils` creare un metodo che analizzi la data specificata secondo il pattern specificato in input, e restituisca una `LocalDate`.

Esercizio 14.n)

Creare una classe `TestDateUtils` per testare la correttezza dei metodi di `DateUtils`.

Esercizio 14.o)

Creare un semplice programma che simuli il lancio di un dado. Eseguendo il programma sarà stampato un numero casuale da 1 a 6. Si utilizzi la documentazione ufficiale per trovare un modo per generare numeri casuali.
Suggerimento: basta un'unica classe con un metodo `main()` contenente un unico statement.

Esercizio 14.p)

Riprendiamo il discorso iniziato nell'esercizio 5.r, implementato nell'esercizio 6.z, e formalizzato nell'esercizio 7.z, come caso d'uso nel caso di studio chiamato *Logos*, introdotto nel paragrafo 5.4.
Nel caso d'uso dell'autenticazione, un utente inseriva uno username e una password per autenticarsi nel sistema. Ma chi ha inserito nel sistema le credenziali per permettere all'utente di autenticarsi con username e password? La risposta la troviamo nel diagramma dei casi d'uso, riportato nuovamente nella figura 14.p.1.

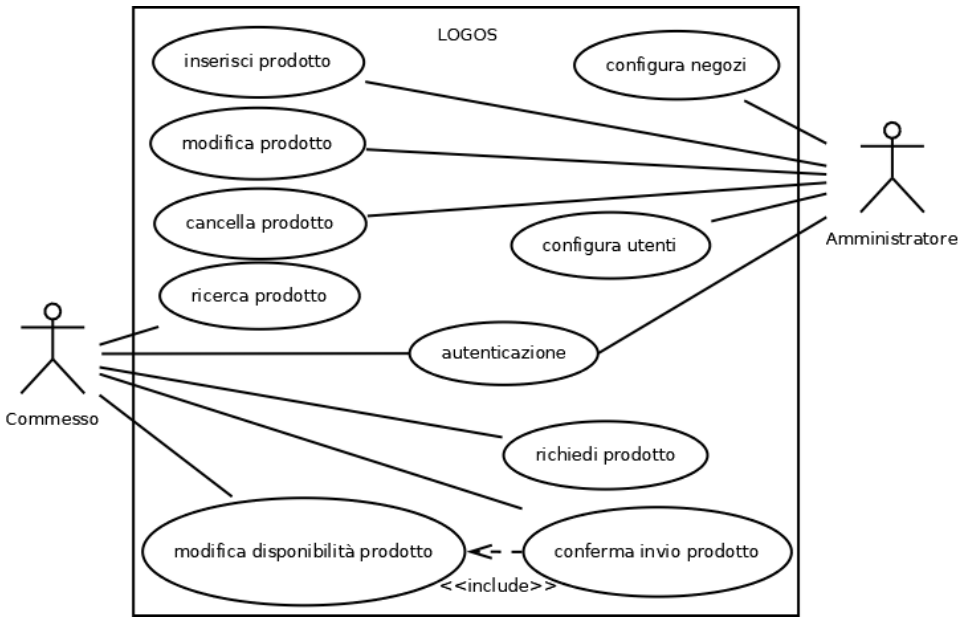

Figura 14.p.1 - Use case diagram aggiornato di Logos.

Il caso d'uso *configura utenti*, è il caso d'uso che questo esercizio richiede di implementare. Anche questo caso d'uso deve essere inteso come un piccolo programma funzionante, indipendentemente da Logos, in modo tale da poterlo eventualmente riutilizzare in altri programmi. Per poter immagazzinare le coppie username e password, utilizzare un file di properties. Tenere presente che oltre a username e password, la classe `Utente` deve avere anche altre proprietà da configurare, come il nome e il ruolo.

Riutilizzare il codice degli esercizi 5.z e 6.z e degli altri esercizi correlati.

Esercizio 14.q)

Tenendo presente la soluzione dell'esercizio precedente, modificare il codice dell'esercizio 6.z ed in particolare la classe `ProfiliUtenti`, in modo tale da caricare gli oggetti `Utente` a partire dal file di properties.

Esercizio 14.r)

Quali tra le seguenti affermazioni sono corrette?

1. Un oggetto `Formatter` permette di immagazzinare coppie chiave-valore come un oggetto `Properties`, senza il bisogno di dover specificare un file da utilizzare per l'immagazzinamento dei dati.
2. Un oggetto `System.out` è di tipo `java.io.PrintStream`.
3. La classe `java.io.PrintStream`. definisce il metodo `printf()` che sfrutta il metodo `format()` della classe `Formatter`.
4. La classe `Formatter` definisce un overload del metodo `format()`.
5. La classe `Formatter` permette di formattare anche in base a oggetti `Locale`.

Esercizio 14.s)

Quali tra le seguenti affermazioni sono corrette riguardo la classe `Preferences`?

1. Un oggetto `Preferences` permette di immagazzinare coppie chiave-valore come un oggetto `Properties`, senza il bisogno di dover specificare un file da utilizzare per l'immagazzinamento dei dati.

2. Un oggetto `Preferences` non usa il metodo `setProperty()`.
3. La classe `Preferences` estende `HashMap`.
4. La classe `Preferences` si istanzia senza poter utilizzare costruttori, ma metodi che restituiscono istanze. Infatti è addirittura una classe astratta.
5. La classe `Preferences` definisce metodi per inserire specifici tipi di dati come valore.

Esercizio 14.t)

Quali tra le seguenti affermazioni sono corrette riguardo le regular expression?

1. La classe `Matcher` definisce il metodo `find()` che ritorna la stringa che coincide con l'espressione booleana di `Matcher`.
2. La classe `Matcher` si istanzia senza poter utilizzare costruttori, ma metodi che restituiscono istanze.
3. Un oggetto `Pattern` definisce i metodi `start()` e `end()`.
4. All'interno di stringhe Java, il simbolo `\` deve essere ripetuto, per evitare che il compilatore lo consideri il prefisso di un carattere di escape.
5. Un "greedy quantifier" è un simbolo che simboleggia la molteplicità di occorrenze coincidenti con un certo pattern.

Esercizio 14.u)

Quali tra le seguenti affermazioni sono corrette riguardo la standardizzazione dei metodi della libreria Date And Time API?

1. I metodi `from` hanno un parametro sempre meno completo rispetto al tipo che devono ritornare.
2. I metodi `of` ritornano istanze di oggetti sulla classe su cui sono invocati.
3. I metodi `plus` e `minus` restituiscono copie dei parametri in input.
4. `with`, `is`, `to` e `at` sono usati solo come prefissi, mai come nomi completi.

Esercizio 14.v)

Quali tra le seguenti affermazioni sono corrette riguardo la geolocalizzazione della

libreria Date And Time API?

1. La classe `ZoneId` astrae il concetto di area geografica che condivide lo stesso orario, ed è solitamente individuata da una coppia del tipo "regione/città".
2. La classe `ZoneOffset` astrae il concetto di fuso orario.
3. La classe `ZoneDateTime` rappresenta la versione localizzata della classe `LocaleDate`.
4. Il metodo `atZone()` della classe `LocalDateTime` restituisce una `ZoneDateTime`.

Esercizio 14.z)

Quali tra le seguenti affermazioni sono corrette riguardo la gestione del codice legacy della libreria Date And Time API?

1. Il metodo `from()` è dichiarato sia nella classe `GregorianCalendar` sia nella classe `Date`.
2. Il metodo `toInstant()` per ottenere un'istanza della classe `Instant`, è dichiarato sia nella classe `Calendar` sia nella classe `Date`.
3. È possibile sostituire la classe `Date` con la classe `Instant`.
4. La classe `GregorianCalendar` può essere sostituita a seconda dei casi con i tipi `ZonedDateTime`, `LocalTime` o `LocalDate`.
5. La classe `TimeZone` può essere sostituita a seconda dei casi con i tipi `ZoneId` o `ZoneOffset`.

Soluzioni degli esercizi capitolo 14

Soluzione 14.a) Package `java.util`*, Vero o Falso:*

1. **Vero**.
2. **Vero**.
3. **Vero**.
4. **Falso**, tutte le "t" non dovrebbero esserci.
5. **Falso**, è valido ma stamperà `false`. Affinché stampi `true` l'espressione si deve modificare in "`\\bb`".
6. **Falso**, il modo in cui vengono immagazzinati i dati persistenti è dipendente dalla piattaforma. Un oggetto `Properties` invece può usare anche file di configurazione in formato XML.
7. **Falso**, la classe `PrintStream` definisce il metodo `printf()`, la classe `Formatter` definisce il metodo `format()`.
8. **Falso**, è un gruppo di caratteri.
9. **Falso**, la classe `Date` non ha un metodo `now()`. La prima riga da sola avrebbe adempiuto al compito richiesto.
10. **Falso**, la classe `SimpleNumberFormat` semplicemente non esiste. Ma esiste la classe `SimpleDateFormat`.

Soluzione 14.b) Date-Time API, Vero o Falso:

1. **Vero**.
2. **Falso**, anche la stessa classe `Instant` definisce un metodo `parse()`.
3. **Falso**, `Duration` non è connessa alla timeline visto che rappresenta un intervallo di tempo compreso tra due `Instant`.
4. **Vero**.
5. **Falso**.
6. **Falso**, la classe `ZonedDate` semplicemente non esiste.
7. **Falso**.
8. **Vero**.
9. **Falso**.
10. **Vero**.

Soluzione 14.c)

Il listato della classe `Traduttore` potrebbe essere come il seguente:

```
import java.util.Locale;
import java.util.ResourceBundle;

public class Traduttore {

    private LinguaEnum lingua;

    private ResourceBundle resources;

    public Traduttore (LinguaEnum lingua) {
        this.lingua = lingua;
        String chiaveLingua = lingua.getChiave();
        Locale locale = new Locale(chiaveLingua);
        resources = ResourceBundle.getBundle(
          "risorse.vocabolario", locale);
    }

    public String traduci(ParoleEnum testo) {
        String traduzione = resources.getString(testo.getChiave());
        return traduzione;
    }

    public void setLingua(LinguaEnum lingua) {
```

```
        this.lingua = lingua;
    }

    public LinguaEnum getLingua() {
        return lingua;
    }
}
```

Si noti che per semplificare il nostro esercizio abbiamo deciso di limitare il numero di parole da tradurre mediante l'enumerazione `ParoleEnum`:

```
public enum ParoleEnum {

    LIBRO("libro"), TEMPO("tempo"), CASA("casa");

    private String chiave;

    private ParoleEnum(String chiave) {
        this.chiave = chiave;
    }

    public String getChiave() {
        return chiave;
    }
}
```

Abbiamo anche limitato il numero di lingue supportate tramite l'enumerazione `LingueEnum`:

```
public enum LinguaEnum {

    ITALIANO("it", "Italiano"), INGLESE("en", "Inglese");

    String chiave;

    String descrizione;

    LinguaEnum(String chiave, String descrizione) {
        this.chiave = chiave;
        this.descrizione = descrizione;
    }

    public String getChiave() {
        return chiave;
    }

    public String toString() {
        return descrizione;
    }
}
```

Inoltre abbiamo creato nella cartella risorse (inserita nella cartella dei sorgenti scaricabili dallo stesso indirizzo da dove avete scaricato questi file: http://www.claudiodesio.com/java9.html) i file di properties che ci servivano: vocabolario_it.properties:

```
libro=libro
casa=casa
tempo=tempo
```

e vocabolario_en.properties:

```
libro=book
casa=home
tempo=time
```

Soluzione 14.d)

Il listato potrebbe essere il seguente:

```
public class TestTraduttore {

    public static void main(String args[]) {
        Traduttore traduttore = new Traduttore(LinguaEnum.INGLESE);
        String parolaTradotta = traduttore.traduci(ParoleEnum.LIBRO);
        System.out.println(parolaTradotta);
    }
}
```

L'output sarà:

```
book
```

potremmo anche usare la classe `Scanner` e rendere interattivo il programma.

Soluzione 14.e)

Il listato potrebbe essere il seguente:

```
import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class StringUtils {

    public static List<String> ricerca(String testo, char iniziale) {
        List<String> list = new ArrayList<>();
        Pattern pattern = Pattern.compile("\\b"+iniziale+"[a-zA-Z]+\\b");
```

```
        Matcher matcher = pattern.matcher(testo);
        while (matcher.find()) {
            list.add(matcher.group());
        }
        return list;
    }
}
```

Soluzione 14.f)

Il listato della classe `TestStringUtils` potrebbe essere il seguente:

```
import java.util.List;

public class TestStringUtils {

    public static void main(String args[]) {
        List<String> list = StringUtils.ricerca(
          "The smile of dawn arrived early May "
          + "she carried a gift from her home "
          + "the night shed a tear to tell her of fear "
          + "and of sorrow and pain she'll never outgrow ", 't');
        for (String string : list) {
            System.out.println(string);
        }
    }
}
```

L'output del codice precedente è

```
the
tear
to
tell
```

Soluzione 14.g)

Il listato potrebbe essere il seguente:

```
import java.time.DayOfWeek;
import java.time.Duration;
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;
import java.time.temporal.ChronoUnit;
import java.time.temporal.TemporalAdjusters;
```

```
public class DateUtils {

    public static long getIntervallo(Instant instant1, Instant instant2,
      ChronoUnit chronoUnit) {
        return chronoUnit.between(instant1, instant2);
    }
}
```

Ma ci sono tante alternative a questa soluzione.

Soluzione 14.h)

Il listato del metodo richiesto è davvero banale:

```
public static long getTempoPassato(Instant instant1, ChronoUnit chronoUnit) {
    return getIntervallo(instant1, Instant.now(), chronoUnit);
}
```

Infatti abbiamo risfruttato il metodo scritto nell'esercizio precedente.

Soluzione 14.i)

Il listato del metodo richiesto è davvero banale:

```
public static String oraEsatta() {
    LocalTime ora = LocalTime.now();
    String oraEsatta = (ora.getHour() + ":" + ora.getMinute() + " "
      + ora.getSecond());
    return oraEsatta;
}
```

Soluzione 14.l)

Il listato del metodo richiesto è anch'esso molto semplice:

```
public static String formattaData(
  LocalDateTime localDateTime, String pattern) throws DateTimeException {
    String formattedDate = null;
    try {
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern(pattern);
        formattedDate = formatter.format(localDateTime);
    } catch (DateTimeException dateTimeException) {
        dateTimeException.printStackTrace();
        throw dateTimeException;
    }
    return formattedDate;
}
```

Si noti che abbiamo gestito l'eccezione solo per stamparne lo stack trace, per poi

rilanciarla. Abbiamo anche inserito la clausola `throws` accanto alla definizione del metodo. In questo modo chi userà questo metodo saprà che potrebbe lanciare l'eccezione nel caso si specifichi un pattern errato.

Soluzione 14.m)

Il listato del metodo richiesto potrebbe essere il seguente:

```
public static LocalDate analizzaData(String data, String pattern)
  throws DateTimeParseException {
    LocalDate localDate = null;
    try {
        localDate = LocalDate.parse(data, DateTimeFormatter.ofPattern(pattern));
    } catch (DateTimeParseException dateTimeParseException) {
    dateTimeParseException.printStackTrace();
        throw dateTimeParseException;
    }
    return localDate;
}
```

Si noti che anche in questo caso abbiamo gestito l'eccezione solo per stamparne lo stack trace, per poi rilanciarla. Abbiamo anche inserito la clausola `throws` accanto alla definizione del metodo. In questo modo chi userà questo metodo saprà che potrebbe lanciare l'eccezione nel caso si specifichi un pattern errato. L'eccezione questa volta è però di tipo `DateTimeParseException`.

Anche in questo caso la soluzione è molto semplice. Si noti che è possibile anche usare altre classi e altri metodi per ottenere gli stessi risultati ottenuti negli ultimi quattro esercizi. La libreria per la gestione delle date e del tempo è davvero semplice e potente.

Soluzione 14.n)

Il listato richiesto potrebbe essere il seguente:

```
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;

public class TestDateUtils {
```

```
    private static final String FORMATO_DATA = "MM/dd/yy  hh:mm a";

    public static void main(String args[]) {
        final String oraEsatta = DateUtils.oraEsatta();
        System.out.println("Sono le: " + oraEsatta);
        Instant duemila = Instant.parse("2000-01-01T00:00:00.00Z");
        Instant duemiladieci = Instant.parse("2010-01-01T00:00:00.00Z");
        long intervalloInGiorni = DateUtils.getIntervallo(
          duemila, duemiladieci, ChronoUnit.DAYS);
        System.out.println("Dal primo gennaio 2000 al primo gennaio 2010 "
          + "sono passati " + intervalloInGiorni + " giorni");
        final long tempoPassatoInMinuti =
          DateUtils.getTempoPassato(duemila, ChronoUnit.MINUTES);
        System.out.println("Dal primo gennaio 2010 ad oggi sono passati "
          + tempoPassatoInMinuti
          + " minuti");
        LocalDateTime localDateTime = LocalDateTime.now();
        final String dataFormattata =
          DateUtils.formattaData(localDateTime, FORMATO_DATA);
        System.out.println("Data formattata: " + dataFormattata);
        LocalDate localDate =
          DateUtils.analizzaData(dataFormattata, FORMATO_DATA);
        System.out.println(localDate);
        // facciamogli lanciare un'eccezione
        localDate = DateUtils.analizzaData(dataFormattata, "ABC");
    }
}
```

Soluzione 14.o)

Ovviamente la classe giusta da utilizzare è la classe `java.util.Random`, e il suo metodo `nextInt()` che prende in input un limite superiore (escluso), e che ha come limite inferiore `0` (incluso). Basta sommargli il valore `1` e il gioco è fatto. Il listato potrebbe essere il seguente:

```
import java.util.Random;

public class LancioDeiDadi {

    public static void main(String args[]) {
        System.out.println("Lancio il dado... "
          + (1 + new Random().nextInt(6)) + "!");
    }
}
```

Soluzione 14.p)

Riutilizziamo le classi `Utente`:

```
package com.claudiodesio.autenticazione;

public class Utente {

    private String nome;
    private String username;
    private String password;

    public Utente(String n, String u, String p) {
        this.nome = n;
        this.username = u;
        this.password = p;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

La classe `Amministratore`:

```
package com.claudiodesio.autenticazione;

public class Amministratore extends Utente {
    public Amministratore (String nome, String username, String password) {
        super(nome, username, password);
    }
}
```

La classe `Commesso`:

```
package com.claudiodesio.autenticazione;

public class Commesso extends Utente{
    public Commesso(String nome, String username, String password) {
        super(nome, username, password);
    }
}
```

Poi modifichiamo la classe `ProfiliUtenti` in modo tale da gestire i profili degli utenti:

```
package com.claudiodesio.autenticazione;

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

public class ProfiliUtenti {

    private static ProfiliUtenti instance;

    private Properties properties;

    private ProfiliUtenti() {
        properties = new Properties();
        try {
            loadProperties();
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public void loadProperties() throws IOException {
        try (FileInputStream inputStream =
          new FileInputStream("config.properties");) {
            properties.load(inputStream);
        }
    }

    public static ProfiliUtenti getInstance() {
        if (instance == null) {
            instance = new ProfiliUtenti();
        }
        return instance;
    }

    public void inserisciUtente(String[] args) throws IOException {
        String ruolo = args[0];
        String nome= args[1];
        String username = args[2];
        String password = args[3];
```

```
        // Se non si specifica il ruolo Amministratore
        // verrà inserito sempre un Commesso
        Utente utente = (ruolo.equals("Amministratore") ?
          new Amministratore(nome, username, password):
          new Commesso(nome, username, password));
        String valore = nome +","+ ruolo+","+ password;
        properties.setProperty(username, valore);
        try (FileOutputStream fos =
          new FileOutputStream("config.properties")) {
            properties.store(fos,"File di configurazione");
        }
        System.out.println("Inserita la proprietà: " + username
          + "=" + valore);
    }
}
```

In particolare abbiamo aggiunto un metodo per aggiungere profili utente, che ci permette di aggiungere nel file di properties una riga la cui chiave è lo username, e il cui valore è una lista con le altre informazioni dell'utente separate da virgole. Infine con la seguente classe che prende da riga di comando dei valori da inserire, viene eseguita l'applicazione:

```
package com.claudiodesio.autenticazione;

import java.io.*;

public class Esercizio14P {

    public static void main(String args[]) throws IOException {
        if (args.length != 4) {
            System.out.println("Specificare ruolo, nome, username, password");
            System.exit(1);
        }
        ProfiliUtenti.getInstance().inserisciUtente(args);
    }
}
```

Eseguendo l'applicazione, per esempio senza specificare argomenti di riga di comando, otterremo il seguente output:

```
java com.claudiodesio.autenticazione.Esercizio14P
Specificare ruolo, nome, username, password
```

invece specificando i seguenti argomenti:

```
java com.claudiodesio.autenticazione.Esercizio14P Amministratore
  Claudio desio xxxxxxxx
Inserita la proprietà: desio=Claudio,Amministratore,xxxxxxxx
```

otterremo che nel file di properties troveremo il seguente contenuto:

```
#File di configurazione
#Sun Mar 25 19:10:31 CEST 2018
desio=Claudio,Amministratore,xxxxxxxx
```

Soluzione 14.q)

Una possibile soluzione risiede nella modifica della classe `ProfiliUtenti` nel seguente modo:

```
package com.claudiodesio.autenticazione;

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

public class ProfiliUtenti {

    private static ProfiliUtenti instance;

    private Properties properties;

    private ProfiliUtenti() {
        properties = new Properties();
        try {
            loadProperties();
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static ProfiliUtenti getInstance() {
        if (instance == null) {
            instance = new ProfiliUtenti();
        }
        return instance;
    }

    public void loadProperties() throws IOException {
        try (FileInputStream inputStream =
          new FileInputStream("config.properties");) {
            properties.load(inputStream);
        }
    }

    public void inserisciUtente(String[] args) throws IOException {
        String ruolo = args[0];
        String nome= args[1];
        String username = args[2];
```

```
        String password = args[3];
        // Se non si specifica il ruolo Amministratore
        // verrà inserito sempre un Commesso
        Utente utente = (ruolo.equals("Amministratore") ?
          new Amministratore(nome, username, password):
          new Commesso(nome, username, password));
        String valore = nome +","+ ruolo+","+ password;
        properties.setProperty(username, valore);
        try (FileOutputStream fos =
          new FileOutputStream("config.properties")) {
            properties.store(fos,"File di configurazione");
        }
        System.out.println("Inserita la proprietà: " + username
          + "=" + valore);
    }

    public Utente getUtente(String username) {
        String valore = (String)properties.getProperty(username);
        if (valore == null) {
            return null;
        }
        String [] campi = valore.split(",");
        Utente utente = (campi[1].equals("Amministratore") ?
          new Amministratore(campi[0], username, campi[2]):
          new Commesso(campi[0], username, campi[2]));
        return utente;
    }
}
```

In particolare abbiamo creato il metodo `getUtente()` che andiamo a richiamare dalla classe `Autenticazione` opportunamente modificata:

```
package com.claudiodesio.autenticazione;

import java.util.Scanner;

public class Autenticazione {

    public void login() {
        boolean autorizzato = false;
        Scanner scanner = new Scanner(System.in);
        do {
            Stampa.richiediUsername();
            String username = scanner.nextLine();
            Utente utente = ProfiliUtenti.getInstance().getUtente(username);
            if (utente != null) {
                Stampa.richiediPassword();
                String password = scanner.nextLine();
                if (verificaPassword(utente, password)) {
                    Stampa.auguraBenvenuto(utente.getNome());
                    autorizzato = true;
```

```
                } else {
                    Stampa.autenticazioneFallita();
                }
            } else {
                Stampa.usernameInesistente();
            }
        } while (!autorizzato);
    }

    private boolean verificaPassword(Utente utente, String password) {
        boolean trovato = false;
        if (password != null) {
            if (password.equals(utente.getPassword())) {
                trovato = true;
            }
        }
        return trovato;
    }
}
```

Infine il metodo `main()` lo abbiamo spostato nella classe `Esercizio14Q`:

```
package com.claudiodesio.autenticazione;

public class Esercizio14Q {
    public static void main(String args[]) {
        Autenticazione autenticazione = new Autenticazione();
        autenticazione.login();
    }
}
```

Eseguendo l'applicazione tutto funziona come prima:

```
Inserisci username.
Pippo
Utente non trovato!
Inserisci username.
desio
Inserisci password.
yyyyyyyy
Autenticazione fallita
Inserisci username.
desio
Inserisci password.
xxxxxxxx
Benvenuto Claudio
```

Soluzione 14.r)

Tutte le affermazioni sono vere.

Soluzione 14.s)

Tutte le affermazioni sono vere tranne la numero 3. Infatti la classe `Preferences` non estende `HashMap`, ma è una classe astratta che estende `Object`. È sufficiente leggere la documentazione delle API Java.

Soluzione 14.t)

Le affermazioni corrette sono le numero 2, 4 e 5. La numero 1 è falsa perché il metodo `find()` non ritorna una stringa ma un booleano (il metodo `find()` viene in fatti usato solitamente come condizione di un ciclo).
La numero 3 invece è falsa perché i metodi `start()` e `end()` non appartengono alla classe `Pattern` ma vengono dichiarati dalla classe `Matcher`.

Soluzione 14.u)

Tutte le affermazioni sono vere tranne la numero 1. Infatti i metodi `from` hanno un parametro sempre più completo rispetto al tipo che devono ritornare.

Soluzione 14.v)

Tutte le affermazioni sono vere tranne la numero 3. Infatti la classe `ZoneDateTime` rappresenta la versione localizzata della classe `LocaleDateTime`, non di `LocalDate`.

Soluzione 14.z)

Tutte le affermazioni sono vere!

Esercizi del capitolo 15

Gestione dei thread

Abbiamo visto nel capitolo 15 come la gestione dei thread sia un argomento molto complesso. Tuttavia gli esercizi che sono presentati di seguito dovrebbero risultare alquanto fattibili.

Esercizio 15.a) Creazione di `Thread`*, Vero o Falso:*

1. Un thread è un oggetto istanziato dalla classe `Thread` o dalla classe `Runnable`.
2. Il multithreading è solitamente una caratteristica dei sistemi operativi e non dei linguaggi di programmazione.
3. In ogni applicazione al runtime esiste almeno un thread in esecuzione.
4. A parte il thread principale, un thread ha bisogno di eseguire codice all'interno di un oggetto la cui classe estende `Runnable` o estende `Thread`.
5. Il metodo `run()` deve essere chiamato dal programmatore per attivare un thread.
6. Il thread corrente non si identifica solitamente con il reference `this`.
7. Chiamando il metodo `start()` su di un thread, questo viene immediatamente eseguito.
8. Il metodo `sleep()` è statico e permette di far dormire per un numero specificato di millisecondi il thread che legge tale istruzione.

9. Assegnare le priorità ai thread è una attività che può produrre risultati diversi su piattaforme diverse.

10. Lo scheduler della JVM non dipende dalla piattaforma su cui viene eseguito.

Esercizio 15.b) Gestione del multi-threading, Vero o Falso:

1. Un thread astrae un processore virtuale che esegue codice su determinati dati.

2. La parola chiave `synchronized` può essere utilizzata sia come modificatore di un metodo sia come modificatore di una variabile.

3. Il monitor di un oggetto può essere identificato con la parte sincronizzata dell'oggetto stesso.

4. Affinché due thread che eseguono lo stesso codice e condividono gli stessi dati non abbiano problemi di concorrenza, è necessario sincronizzare il codice comune.

5. Si dice che un thread ha il lock di un oggetto se entra nel suo monitor.

6. I metodi `wait()`, `notify()` e `notifyAll()` rappresentano il principale strumento per far comunicare più thread.

7. I metodi `suspend()` e `resume()` sono attualmente deprecati.

8. Il metodo `notityAll()`, invocato su di un certo oggetto `o1`, risveglia dallo stato di pausa tutti i thread che hanno invocato `wait()` sullo stesso oggetto. Tra questi verrà eseguito quello che era stato fatto partire per primo con il metodo `start()`.

9. Il deadlock è una condizione di errore bloccante generata da due thread che stanno in reciproca dipendenza in due oggetti sincronizzati.

10. Se un thread `t1` esegue il metodo `run()` nell'oggetto `o1` della classe `C1`, e un thread `t2` esegue il metodo `run()` nell'oggetto `o2` della stessa classe `C1`, la parola chiave `synchronized` non serve a niente.

Esercizio 15.c)

Simulare con del codice funzionante la seguente situazione con quanto appreso in questo capitolo.

Un gruppo di 10 persone è all'osservatorio astronomico per ammirare il passaggio di una cometa sfruttando il potente telescopio messo a disposizione dalla struttura. Solo una persona alla volta può usare il telescopio, e i responsabili concedono solo tre minuti a persona per completare l'osservazione. Non esiste una fila, i partecipanti accederanno al telescopio in maniera casuale. Ogni partecipante quindi attraverserà degli stati:

- lo stato "In attesa" quando starà aspettando il suo turno. Dura un tempo indefinito, che dipende da quando inizierà il turno del partecipante;
- lo stato "Osservazione" quando starà osservando la cometa tramite il telescopio. Dura esattamente 3 minuti (ma è possibile abbreviare questo tempo per eseguire l'esercizio);
- lo stato "Finito" quando il turno è finito.

Gli stati possono essere caratterizzati da delle stampe significative.

Esercizio 15.d)

Simulare con del codice funzionante la seguente situazione con quanto appreso in questo capitolo.
Supponiamo di trovarci allo sportello del comune che rilascia le carte d'identità. Supponiamo che ci siano 10 richiedenti in fila pronti per richiedere il documento. Quando arriva il proprio turno, sarà consegnato al richiedente un modulo da compilare. Per non bloccare la fila, il richiedente successivo potrà nel frattempo richiedere il servizio allo stesso sportello. Quindi a tutti i richiedenti sarà consegnato il modulo da compilare, la fila sarò molto veloce, e in parallelo diverse persone saranno occupate a compilare il modulo. Ogni richiedente potrebbe metterci un tempo variabile da 1 a 10 secondi per compilare il modello, il primo che finirà (indipendentemente dalla sua posizione nella fila iniziale) potrà richiedere la stampa della propria carta d'identità. Questa sarà stampata in 3 secondi in ogni caso.
Anche in questo caso usare delle stampe significative per rendere l'esecuzione del programma auto esplicativa.

Esercizio 15.e)

Consideriamo le seguenti classi:

```
import java.util.ArrayList;

class RunnableArrayList extends ArrayList implements Runnable {
```

```
    public void run(String stringa) {
        System.out.println("Nel metodo run(): " + stringa);
    }
}

public class Esercizio15E {

    public static void main(String args[]) {
        RunnableArrayList g = new RunnableArrayList();
        Thread t = new Thread(g);
        t.start();
    }
}
```

Se eseguiamo la classe `Esercizio15E`, quale sarà l'output?

1. `Nel metodo run(): null`
2. `Nel metodo run():`
3. Il codice viene interrotto con un'eccezione nella classe `Esercizio15E`.
4. Il codice viene interrotto con un'eccezione nella classe `RunnableArrayList`.
5. Nessun output. Il codice non compila per un errore nella classe `Esercizio15E`.
6. Nessun output. Il codice non compila per un errore nella classe `RunnableArrayList`.

Esercizio 15.f)

Quali delle seguenti affermazioni sono corrette:

1. Una classe, per creare oggetti che abbiano un monitor, deve estendere `Thread`.
2. Una classe, per creare oggetti che abbiano un monitor, deve implementare `Runnable`.
3. Quando "un thread entra nel monitor dell'oggetto" allora possiede il lock di tale oggetto. Questo significa che nessun altro thread può entrare nel monitor di quell'oggetto.
4. La classe `Monitor` è definita dai metodi sincronizzati di una classe.

Esercizio 15.g)

Quali delle seguenti affermazioni sono corrette?

1. La classe `Thread` estende `Runnable`.
2. `Runnable` è una functional interface.
3. La classe `Thread` definisce un metodo `run()` vuoto.
4. La classe `Thread` definisce i metodi `wait()` e `notify()`.

Esercizio 15.h)

Creare una classe `ContoAllaRovescia` che una volta attivata fa partire un conto alla rovescia da 10 a 0 prima di terminare.

Esercizio 15.i)

Quali delle seguenti affermazioni sono corrette:

1. Per istanziare un thread basta usare i suoi costruttori.
2. Specificare la priorità dei thread non rappresenta un modo efficace per decidere l'ordine di esecuzione di più thread.
3. Per eseguire il metodo `run()` definito in una classe che estende `Thread` basta chiamare il suo metodo `run()`.
4. Per eseguire un thread bisogna invocare il metodo `start()` della classe `Object`.
5. In ogni programma ci sono almeno tre thread in esecuzione.

Esercizio 15.l)

Creare un programma che simuli la seguente situazione. Tramite comandi interattivi (corri, cammina, fermati e basta) letti dalla classe `Scanner`, l'utente reciterà la parte dell'allenatore di un corridore virtuale, dando dei comandi al volo mentre il corridore si allena.
Creare quindi una classe `Corridore` che estende `Thread`. Essa deve dichiarare i metodi: `corri()` che gli permetterà di simulare una corsa, `cammina()` che gli permetterà di simulare una camminata, `fermati()` che gli permetterà di simulare l'azione del fermarsi, e `basta()` che farà terminare l'allenamento (e il programma).

Suggerimento: possiamo sfruttare variabili booleane per gestire un ciclo infinito che gestisce il ciclo dell'allenamento.

Esercizio 15.m)

È meglio implementare `Runnable` o estendere la classe `Thread`? Scegliere tutte le affermazioni corrette.

1. Implementando `Runnable` che è un'interfaccia, possiamo anche estendere altre classi.
2. Estendendo la classe `Thread` possiamo anche implementare altre interfacce.
3. Dal punto di vista dell'astrazione dei dati, un oggetto di tipo `Thread` non dovrebbe possedere variabili private che rappresentano i dati da gestire.
4. Un oggetto `Runnable` che implementa il metodo `run()`, può essere passato in input al costruttore di un oggetto `Thread`. Se viene invocato su quest'ultimo il metodo `start()`, viene attivato il thread che esegue il metodo `run()`.

Esercizio 15.n)

Quali delle seguenti affermazioni sono corrette?

1. Con il time slicing o round-robin scheduling un thread può trovarsi in esecuzione solo per un certo periodo di tempo.
2. Il time slicing o round-robin scheduling è il comportamento di default della maggior parte dei sistemi Linux.
3. Il preemptive scheduling è il comportamento di default della maggior parte dei sistemi Linux.
4. Con il preemptive scheduling la priorità è un elemento più deterministico rispetto al caso del round robin. Infatti lanciati due thread in contemporanea, il thread a maggiore priorità entrerà nello stato di esecuzione, e ne uscirà solo quando avrà terminato il suo lavoro, oppure nel caso sia chiamato su di esso un metodo come `wait()` o `suspend()`, oppure si metta in attesa di risorse esterne (come nel caso di attesa di risorse di input-output).

Esercizio 15.o)

Quali delle seguenti affermazioni sono corrette?

1. Il modificatore `volatile` si può usare solo su metodi e variabili.
2. Dichiarare `volatile` una variabile d'istanza, implica che saranno considerate `volatile` anche tutte le altre variabili della stessa istanza che vengono usate dal medesimo thread.
3. Per una variabile `volatile`, tutti gli accessi in lettura e scrittura sono atomici.
4. Per una variabile intera non `volatile`, tutti gli accessi in lettura e scrittura sono atomici.

Esercizio 15.p)

Quali delle seguenti affermazioni sono corrette?

1. La parola chiave `synchronized` permette di dichiarare metodi atomici.
2. Il monitor di un oggetto è costituito dalla parte dell'oggetto sincronizzata. Quindi se in un certo istante, un thread sta eseguendo codice all'interno di uno dei metodi sincronizzati di tale oggetto, qualsiasi altro thread che vuole accedere al codice di uno qualsiasi dei metodi sincronizzati dello stesso oggetto, dovrà attendere che il primo thread termini di eseguire codice sincronizzato.
3. È possibile utilizzare blocchi sincronizzati per rendere sincronizzati solo determinate righe di codice. Questo rappresenta una soluzione più flessibile della sincronizzazione.
4. È possibile utilizzare `synchronized` come modificatore solo con i metodi.
5. È possibile dichiarare `synchronized` il metodo `run()`.

Esercizio 15.q)

Quali delle seguenti affermazioni sono corrette?

1. I metodi `wait()`, `notify()` e `notifyAll()` sono definiti nella classe `Thread`.
2. Se un thread incontra il metodo `wait()` abbandona il monitor dell'oggetto di cui stava eseguendo codice.

3. Il metodo `notifyAll()` fa ripartire tutti i thread che avevano eseguito il metodo `wait()`.
4. I metodi `suspend()` e `resume()` sono definiti nella classe `Thread`.

Esercizio 15.r)

Rendere immutabile la seguente classe:

```
import java.util.Date;

public final class Esercizio15R {

    private final Integer intero;

    private final Date date;

    public Esercizio15R(Integer intero, Date date) {
        this.intero = intero;
        this.date = (Date)date.clone();
    }

    public final Date getStringBuilder() {
        return (Date)date.clone();
    }

    public final Integer getIntero() {
        return intero;
    }
}
```

Esercizio 15.s)

Quali delle seguenti affermazioni sono corrette riguardo i package `java.util.concurrent.atomic` e `java.util.concurrent.locks`?

1. `ReentrantLock` è una classe che, se istanziata, può sostituire l'utilizzo di un blocco sincronizzato. Però bisogna utilizzare obbligatoriamente un blocco `try-catch-finally`.
2. Il concetto di *fairness* degli oggetti di tipo `ReentrantLock` permette alla JVM di creare una gearchia di priorità di esecuzione dei thread, basata sul tempo di creazione del `ReentrantLock`.
3. Un oggetto di tipo `Lock` può specificare un timeout per uscire da un blocco sincronizzato.

4. L'interfaccia `AtomicInteger` definisce metodi atomici che compiono più di un'operazione.

Esercizio 15.t)

Considerato il seguente snippet:

```
Callable<String> callable = new Callable<>() {public void call(){}};
Future<String> future = Executors.newFixedThreadPool(3).start(callable);
String result = future.get();
```

Che messaggio restituirà il compilatore?

Esercizio 15.u)

Realizzare una sveglia con le classi `Timer` e `TimerTask` del package `java.util`. In particolare deve essere possibile passare da riga di comando un numero che rappresenterà il numero di secondi che devono passare affinché "suoni" la sveglia.

Esercizio 15.v)

Quali delle seguenti affermazioni sono corrette?

1. Lo statement:

   ```
   new Semaphore().acquire();
   ```

   permette ad un semaforo di acquisire il lock su un oggetto.

2. Il metodo `tryAcquire()` permette di specificare un timeout.

3. `Semaphore` è un'interfaccia.

4. I *permits* rappresentano una variabile d'istanza di `Semaphore`.

Esercizio 15.z)

Quali delle seguenti affermazioni sono corrette?

1. Istanziato un oggetto `cyclicBarrier` di tipo `CyclicBarrier`, lo statement:

   ```
   cyclicBarrier.await();
   ```

   equivale a chiamare il metodo `wait()` sull'oggetto che legge questa istruzione.

2. Il costruttore di `CyclicBarrier` permette di specificare il numero di thread che devono "accumularsi" in un certo punto di codice prima di essere rilasciati.
3. Il metodo `signalAll()` di `CyclicBarrier` è equivalente al metodo `notifyAll()` di `Object`.
4. L'utilizzo di `CyclicBarrier` può sempre sostituire l'utilizzo di `wait()`, `notify()` e `notifyAll()`.

Soluzioni degli esercizi del capitolo 15

Soluzione 15.a) Creazione di `thread`*, Vero o Falso:*

1. **Falso**, `Runnable` è un'interfaccia.
2. **Vero**.
3. **Vero**, il cosiddetto thread "main".
4. **Vero**.
5. **Falso**, il programmatore può invocare il metodo `start()` e lo scheduler invocherà il metodo `run()`.
6. **Vero**.
7. **Falso**.
8. **Vero**.
9. **Vero**.
10. **Falso**.

Soluzione 15.b) Gestione del multithreading, Vero o Falso:

1. **Vero**.
2. **Falso**.

3. Vero.

4. Falso.

5. Vero.

6. Vero.

7. Vero.

8. Falso, il primo thread che partirà sarà quello a priorità più alta.

9. Vero.

10. Vero.

Soluzione 15.c)

Per prima cosa astraiamo il concetto di stato di cui si è parlato nel quesito 15.c, tramite un'enumerazione. Creiamo anche il messaggio da stampare per ogni stato:

```
package com.claudiodesio.osservatorio.dati;

public enum Stato {

    IN_ATTESA("\"Sono in attesa...\""),
    OSSERVAZIONE("\"Tocca a me... che meraviglia!\""),
    FINITO("\"Ho finito.\"");

    private String messaggio;

    private Stato(String messaggio) {
        this.messaggio = messaggio;
    }

    public String getMessaggio() {
        return messaggio;
    }
}
```

Poi creiamo una semplice classe `Partecipante`, che ha tra le sue caratteristiche proprio un oggetto `Stato`. Tutti i partecipanti hanno un nome e un telescopio a cui fanno riferimento per l'osservazione. La nostra classe estende `Thread`, e il suo metodo `run()` definisce l'azione dell'osservazione.

```
package com.claudiodesio.osservatorio.dati;

public class Partecipante extends Thread {
```

```
    private final String nome;

    private Stato stato;

    private final Telescopio telescopio;

    public Telescopio getTelescopio() {
        return telescopio;
    }

    public Partecipante(String nome, Telescopio telescopio) {
        this.nome = nome;
        this.telescopio = telescopio;
        this.setStato(Stato.IN_ATTESA);
        stato();
    }

    public Stato getStato() {
        return stato;
    }

    public void setStato(Stato stato) {
        this.stato = stato;
    }

    public String getNome() {
        return nome;
    }

    @Override
    public void run() {
        telescopio.permettiOsservazione(this);
    }

    public void stato() {
        System.out.println(nome + " dice: " + stato.getMessaggio());
    }
}
```

Si noti che, quando viene creato un `Partecipante`, lo stato viene impostato a `IN_ATTESA`.

Poi creiamo la classe più importante, ovvero quella che astrae il concetto di `Telescopio`:

```
package com.claudiodesio.osservatorio.dati;

public class Telescopio {

    public synchronized void permettiOsservazione(Partecipante partecipante) {
        partecipante.setStato(Stato.OSSERVAZIONE);
```

```
            partecipante.stato();
            try {
                Thread.sleep(3000);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
            partecipante.setStato(Stato.FINITO);
            partecipante.stato();
        }
    }
```

Nel suo metodo di business sincronizzato, si limita a passare da uno stato ad un altro dopo una pausa di 3 secondi.
Infine, segue una classe di test:

```
package com.claudiodesio.osservatorio.test;

import com.claudiodesio.osservatorio.dati.Partecipante;
import com.claudiodesio.osservatorio.dati.Telescopio;

public class Osservazione {

    public static void main(String args[]) {
        Telescopio telescopio = new Telescopio();
        Partecipante[] partecipanti = getPartecipanti(telescopio);
        for (Partecipante partecipante : partecipanti) {
            partecipante.start();
        }
    }

    private static Partecipante[] getPartecipanti(Telescopio telescopio) {
        Partecipante[] partecipanti = {
            new Partecipante("Ciro", telescopio),
            new Partecipante("Gianluca", telescopio),
            new Partecipante("Pierluigi", telescopio),
            new Partecipante("Gigi", telescopio),
            new Partecipante("Nicola", telescopio),
            new Partecipante("Pino", telescopio),
            new Partecipante("Maurizio", telescopio),
            new Partecipante("Raffaele", telescopio),
            new Partecipante("Fabio", telescopio),
            new Partecipante("Vincenzo", telescopio)};
        return partecipanti;
    }
}
```

L'output è il seguente (che cambia ad ogni lancio):

```
Ciro dice: "Sono in attesa..."
Gianluca dice: "Sono in attesa..."
```

```
Pierluigi dice: "Sono in attesa..."
Gigi dice: "Sono in attesa..."
Nicola dice: "Sono in attesa..."
Pino dice: "Sono in attesa..."
Maurizio dice: "Sono in attesa..."
Raffaele dice: "Sono in attesa..."
Fabio dice: "Sono in attesa..."
Vincenzo dice: "Sono in attesa..."
Ciro dice: "Tocca a me... che meraviglia!"
Ciro dice: "Ho finito."
Vincenzo dice: "Tocca a me... che meraviglia!"
Vincenzo dice: "Ho finito."
Maurizio dice: "Tocca a me... che meraviglia!"
Maurizio dice: "Ho finito."
Pino dice: "Tocca a me... che meraviglia!"
Pino dice: "Ho finito."
Fabio dice: "Tocca a me... che meraviglia!"
Fabio dice: "Ho finito."
Gigi dice: "Tocca a me... che meraviglia!"
Gigi dice: "Ho finito."
Nicola dice: "Tocca a me... che meraviglia!"
Nicola dice: "Ho finito."
Raffaele dice: "Tocca a me... che meraviglia!"
Raffaele dice: "Ho finito."
Pierluigi dice: "Tocca a me... che meraviglia!"
Pierluigi dice: "Ho finito."
Gianluca dice: "Tocca a me... che meraviglia!"
Gianluca dice: "Ho finito."
```

Soluzione 15.d)

Creiamo una classe `TimeUtils` che definisce un metodo d'utilità per generare un numero casuale tra 5 e 10:

```
package com.claudiodesio.sportello.dati;

import java.util.*;

public class TimeUtils {

    private static final Random RANDOM = new Random();

    public static int getNumeroRandom() {
        return (RANDOM.nextInt(6) + 5);
    }
}
```

La classe `Richiedente` estende `Thread`:

```
package com.claudiodesio.sportello.dati;

public class Richiedente extends Thread {

    private final String nome;

    public Richiedente(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }

    @Override
    public void run() {
        Sportello.getInstance().gestisciRichiesta(this);
    }

    @Override
    public String toString() {
        return nome;
    }
}
```

La classe `Stampante` sarà usata per stampare i documenti:

```
package com.claudiodesio.sportello.dati;

public class Stampante {

    private static Stampante instance;

    private Stampante() {
    }

    public static Stampante getInstance() {
        if (instance == null) {
            instance = new Stampante();
        }
        return instance;
    }

    public synchronized void stampa(Richiedente richiedente) {
        System.out.println("Stampa carta d'identità di " + richiedente
          + " in corso...");
        try {
            wait(3000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
```

```
        System.out.println("Stampa completata! " + richiedente
          + " grazie e arrivederci!");
    }
}
```

La classe `Sportello` è la classe chiave:

```
package com.claudiodesio.sportello.dati;

public class Sportello {

    private final Stampante stampante;
    private static Sportello instance;

    public synchronized static Sportello getInstance() {
        if (instance == null) {
            instance = new Sportello();
        }
        return instance;
    }

    private Sportello() {
        stampante = Stampante.getInstance();
    }

    public synchronized void gestisciRichiesta(Richiedente richiedente) {
        System.out.println("Buongiorno " + richiedente);
        System.out.println("Impiegato dice: \"Prego compili il modulo "
          + richiedente + "\"");
        compilaModulo(richiedente);
        stampante.stampa(richiedente);
        System.out.println(richiedente + " dice: \"Grazie a lei!\"");
    }

    private synchronized void compilaModulo(Richiedente richiedente) {
        System.out.println("Richiedente " + richiedente
          + " dice: \"OK lo compilo subito ma...\"");
        final int attesa = TimeUtils.getNumeroRandom();
        try {
            System.out.println("...mi servono " + attesa + " minuti...");
            wait(attesa * 1000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        System.out.println("Richiedente " + richiedente
          + " dice: \"...ho compilato il modulo!\"");
    }
}
```

Segue la classe di test:

```
package com.claudiodesio.sportello.test;

import com.claudiodesio.sportello.dati.Richiedente;

public class TestSportello {

    public static void main(String args[]) {
        final Richiedente[] richiedenti = getRichiedenti();
        for (Richiedente richiedente : richiedenti) {
            richiedente.start();
        }
    }

    private static Richiedente[] getRichiedenti() {
        Richiedente[] partecipanti = {
            new Richiedente("Ciro"),
            new Richiedente("Mario"),
            new Richiedente("Massimo"),
            new Richiedente("Chicco"),
            new Richiedente("Enrico"),
            new Richiedente("Lorenzo"),
            new Richiedente("Emanuele"),
            new Richiedente("Cosimo"),
            new Richiedente("Alessandro"),
            new Richiedente("Salvatore")};
        return partecipanti;
    }
}
```

Infine, segue l'output (che cambia sempre):

```
Buongiorno Ciro
Impiegato dice: "Prego compili il modulo Ciro"
Richiedente Ciro dice: "OK lo compilo subito ma..."
...mi servono 9 minuti...
Buongiorno Salvatore
Impiegato dice: "Prego compili il modulo Salvatore"
Richiedente Salvatore dice: "OK lo compilo subito ma..."
...mi servono 9 minuti...
Buongiorno Cosimo
Impiegato dice: "Prego compili il modulo Cosimo"
Richiedente Cosimo dice: "OK lo compilo subito ma..."
...mi servono 7 minuti...
Buongiorno Mario
Impiegato dice: "Prego compili il modulo Mario"
Richiedente Mario dice: "OK lo compilo subito ma..."
...mi servono 7 minuti...
Buongiorno Chicco
Impiegato dice: "Prego compili il modulo Chicco"
Richiedente Chicco dice: "OK lo compilo subito ma..."
...mi servono 5 minuti...
```

```
Buongiorno Alessandro
Impiegato dice: "Prego compili il modulo Alessandro"
Richiedente Alessandro dice: "OK lo compilo subito ma..."
...mi servono 9 minuti...
Buongiorno Enrico
Impiegato dice: "Prego compili il modulo Enrico"
Richiedente Enrico dice: "OK lo compilo subito ma..."
...mi servono 8 minuti...
Buongiorno Massimo
Impiegato dice: "Prego compili il modulo Massimo"
Richiedente Massimo dice: "OK lo compilo subito ma..."
...mi servono 8 minuti...
Buongiorno Lorenzo
Impiegato dice: "Prego compili il modulo Lorenzo"
Richiedente Lorenzo dice: "OK lo compilo subito ma..."
...mi servono 5 minuti...
Buongiorno Emanuele
Impiegato dice: "Prego compili il modulo Emanuele"
Richiedente Emanuele dice: "OK lo compilo subito ma..."
...mi servono 9 minuti...
Richiedente Lorenzo dice: "...ho compilato il modulo!"
Stampa carta d'identitÃ  di Lorenzo in corso...
Stampa completata! Lorenzo grazie e arrivederci!
Lorenzo dice: "Grazie a lei!"
Richiedente Massimo dice: "...ho compilato il modulo!"
Stampa carta d'identitÃ  di Massimo in corso...
Stampa completata! Massimo grazie e arrivederci!
Massimo dice: "Grazie a lei!"
Richiedente Enrico dice: "...ho compilato il modulo!"
Stampa carta d'identitÃ  di Enrico in corso...
Stampa completata! Enrico grazie e arrivederci!
Enrico dice: "Grazie a lei!"
Richiedente Mario dice: "...ho compilato il modulo!"
Stampa carta d'identitÃ  di Mario in corso...
Stampa completata! Mario grazie e arrivederci!
Mario dice: "Grazie a lei!"
Richiedente Cosimo dice: "...ho compilato il modulo!"
Stampa carta d'identitÃ  di Cosimo in corso...
Stampa completata! Cosimo grazie e arrivederci!
Cosimo dice: "Grazie a lei!"
Richiedente Chicco dice: "...ho compilato il modulo!"
Stampa carta d'identitÃ  di Chicco in corso...
Stampa completata! Chicco grazie e arrivederci!
Chicco dice: "Grazie a lei!"
Richiedente Emanuele dice: "...ho compilato il modulo!"
Stampa carta d'identitÃ  di Emanuele in corso...
Stampa completata! Emanuele grazie e arrivederci!
Emanuele dice: "Grazie a lei!"
Richiedente Salvatore dice: "...ho compilato il modulo!"
Stampa carta d'identitÃ  di Salvatore in corso...
Stampa completata! Salvatore grazie e arrivederci!
```

```
Salvatore dice: "Grazie a lei!"
Richiedente Alessandro dice: "...ho compilato il modulo!"
Stampa carta d'identitÃ  di Alessandro in corso...
Stampa completata! Alessandro grazie e arrivederci!
Alessandro dice: "Grazie a lei!"
Richiedente Ciro dice: "...ho compilato il modulo!"
Stampa carta d'identitÃ  di Ciro in corso...
Stampa completata! Ciro grazie e arrivederci!
Ciro dice: "Grazie a lei!"
```

Soluzione 15.e)

La risposta corretta è la numero 6, in quanto la classe `RunnableArrayList` non implementa correttamente l'interfaccia `Runnable`, il cui metodo `run()` non prende nessun tipo di parametri in input. Infatti l'output della compilazione è il seguente (sono presenti anche due warning relativi all'utilizzo e alla dichiarazione di raw type):

```
Esercizio15E.java:3: warning: [rawtypes] found raw type: ArrayList
class RunnableArrayList extends ArrayList implements Runnable {
                                ^
  missing type arguments for generic class ArrayList<E>
  where E is a type-variable:
    E extends Object declared in class ArrayList
Esercizio15E.java:3: error: RunnableArrayList is not abstract and does
  not override abstract method run() in Runnable
class RunnableArrayList extends ArrayList implements Runnable {
^
Esercizio15E.java:3: warning: [serial] serializable class
  RunnableArrayList has no definition of serialVersionUID
class RunnableArrayList extends ArrayList implements Runnable {
^
1 error
2 warnings
```

Soluzione 15.f)

Solo la risposta 3 è corretta, tutte le altre no.
In particolare, riguardo la risposta 4, la classe `Monitor` non esiste.

Soluzione 15.g)

Le risposte corrette sono la numero 2 e la numero 3.
La numero 1 è falsa perché la classe `Thread` implementa `Runnable`. Infatti quest'ultima è un'interfaccia e in quanto tale va implementata e non estesa da un'altra classe.

La numero 4 è sbagliata visto che i metodi `wait()` e `notify()` sono dichiarati nella classe `Object`.

Soluzione 15.h)

La classe `ContoAllaRovescia` potrebbe essere la seguente:

```
public class ContoAllaRovescia {

    public void attiva(int secondi) throws InterruptedException {
        for (int i = secondi; i > 0; i--) {
            System.out.println(i);
            Thread.sleep(1000);
        }
        System.out.println("Tempo scaduto!");
    }
}
```

Poi possiamo creare una classe che la testi:

```
public class Esercizion15H {
    public static void main(String args[]) throws Exception {
        ContoAllaRovescia contoAllaRovescia = new ContoAllaRovescia();
        int secondi = 10;
        if (args.length > 0) {
            try {
                secondi = Integer.parseInt(args[0]);
            }
            catch (Exception exc) {
                System.out.println("L'input deve essere un numero "
                  + "intero positivo, usiamo il valore di default 10...");
            }
        }
        contoAllaRovescia.attiva(secondi);
    }
}
```

Eseguendo quest'ultima senza specificare parametri otterremo il seguente output:

```
10
9
8
7
6
5
4
3
2
1
Tempo scaduto!
```

Se passiamo il parametro `3` allora l'output sarà limitato a:

```
3
2
1
Tempo scaduto!
```

Se invece specifichiamo una lettera (supponiamo `F`) avremo il seguente output:

```
L'input deve essere un numero intero positivo, usiamo il valore di
  default 10...
10
9
8
7
6
5
4
3
2
1
Tempo scaduto!
```

Infine, specificando un numero negativo oppure `0`, otterremo direttamente il seguente output:

```
Tempo scaduto!
```

Soluzione 15.i)

Le risposte corrette sono le numero 1, 2 e 3. In particolare la 3 non asserisce che il metodo `run()` viene eseguito in un thread a parte, altrimenti sarebbe stata errata. Il metodo `run()`, rimane comunque un metodo, e quindi è invocabile come qualsiasi altro metodo. L'affermazione numero 4 è errata perché il metodo `start()` è dichiarato dalla classe `Thread`, e non dalla classe `Object`. L'affermazione numero 5 è errata perché la frase giusta sarebbe: "in ogni programma c'è almeno un thread in esecuzione".

Soluzione 15.l)

Le specifiche dell'esercizio 15.l sono volutamente non troppo dettagliate, per lasciare più spazio alla fantasia del lettore. Quindi in questo caso un'eventuale soluzione può differire davvero molto dalla soluzione che presentiamo di seguito.
Per esempio potremmo implementare la classe `Corridore` nel seguente modo:

```
public class Corridore extends Thread {

    private boolean viaLibera;

    private boolean inAzione;

    private int gap;

    public Corridore() {
        inAzione = true;
        gap = 1000;
    }

    public void run() {
        while (inAzione) {
            try {
                Thread.sleep(gap);
                if (viaLibera) {
                    System.out.println("|");
                    Thread.sleep(gap);
                    System.out.println("  |");
                }
            } catch (InterruptedException exc) {
                assert false;
            }
        }
    }

    public void inizia() {
        start();
    }

    public void corri() {
        System.out.println("Ok, vado...");
        gap = 400;
        viaLibera = true;
    }

    public void fermati() {
        System.out.println("Ok, mi fermo.");
        System.out.println("| |");
        viaLibera = false;
    }

    public void cammina() {
        System.out.println("Ok, mi riposo un po'...");
        gap = 1000;
        viaLibera = true;
    }

    public void basta() {
```

```
            System.out.println("Meno male, non ce la facevo più...");
            inAzione = false;
        }
    }
```

Il nucleo della logica di business si trova proprio nel metodo `run()`, che con un ciclo che si basa sulla variabile `inAzione`, stampa con dei simboli | (si legge "pipe" in inglese) una trama che assomiglia a dei passi. Si noti che quando si deve correre, la variabile `gap`, che rappresenta l'attesa da un punto all'altro, è di 400 millisecondi, mentre quando si deve camminare il gap viene allungato ad un secondo. A seconda che sia chiamato il metodo `corri()` o il metodo `cammina()` quindi, questi passi saranno più veloci o meno veloci.

Ma vediamo ora come potremmo implementare la classe principale `Esercizio15L`:

```
import java.util.Scanner;

public class Esercizio15L {
    public static void main(String args[]) {
        Corridore corridore = new Corridore();
        corridore.inizia();
        Scanner scanner = new Scanner(System.in);
        boolean cicla = true;
        System.out.println(
           "Ciao allenatore, il corridore è a tua disposizione!");
        System.out.println("Scrivi i comandi e batti invio");
        System.out.println("(corri, cammina, fermati, basta)");
        while (cicla) {

            String comando = scanner.nextLine();
            switch (comando) {
                case"corri":
                    corridore.corri();
                break;
                case"cammina":
                    corridore.cammina();
                break;
                case"fermati":
                    corridore.fermati();
                break;
                case"basta":
                    corridore.basta();
                    cicla = false;
                break;

                default:
                break;
            }
```

```
            }
            try {
                Thread.sleep(2000);
            }
            catch (Exception exc) {
                assert false;
            }
            System.out.println("Fine allenamento");
        }
    }
```

Il codice non differisce molto da altro che abbiamo già visto tra questi esercizi. Tramite un'oggetto scanner vengono letti i comandi, che si traducono in chiamate ai metodi di `Corridore`.

Se eseguiamo quest'ultima classe potremmo ottenere il seguente output, che non rende bene l'idea se non si vede in azione "dal vivo" (meglio eseguire le classi già pronte dalla cartella Codice\capitolo_15\esercizi\15.l del file degli esercizi che avete probabilmente già scaricato insieme a questi esercizi all'indirizzo http://www.claudiodesio.com/java9.html):

```
Ciao allenatore, il corridore è a tua disposizione!
Scrivi i comandi e batti invio
(corri, cammina, fermati, basta)
corri
Ok, vado...
|
  |
|
  |
|
  |
|
  |
c|
a  |
mmi|
na  |
|

Ok, mi riposo un po'...
  |
|
  |
|
  |
ferm|
ati  |
```

```
Ok, mi fermo.
| |
corri
Ok, vado...
|
  |
|
  |
|
  |
ca|
  |
mm|
ina  |
|
  |
|

Ok, mi riposo un po'...
  |
|
  |
|
fe  |
rmat|
i
Ok, mi fermo.
| |
basta
Meno male, non ce la facevo più...
Fine allenamento
```

Si noti che, i comandi inseriti, si estendono spesso su più righe, perché intanto il programma sta stampando "i passi".

Soluzione 15.m)

Tutte le affermazioni sono corrette tranne la numero 2. Infatti anche implementando l'interfaccia `Runnable` si ha la possibilità di implementare altre interfacce.

Soluzione 15.n)

Tutte le affermazioni sono corrette tranne la numero 3. Infatti il preemptive scheduling è il comportamento di default della maggior parte dei sistemi Unix.

Soluzione 15.o)

Tutte le affermazioni sono corrette tranne la numero 1. Infatti il modificatore `volatile`, si può usare solo su variabili d'istanza.

Soluzione 15.p)

Tutte le affermazioni sono corrette.

Soluzione 15.q)

Tutte le affermazioni sono corrette tranne la numero 1. Infatti metodi `wait()`, `notify()` e `notifyAll()` sono definiti della classe `Object`.

Soluzione 15.r)

Una possibile soluzione è la seguente:

```
import java.util.Date;

public final class Esercizio15R {

    private final Integer intero;

    private final Date date;

    public Esercizio15R(Integer intero, Date date) {
        this.intero = intero;
        this.date = (Date)date.clone();
    }

    public final Date getStringBuilder() {
        return (Date)date.clone();
    }

    public final Integer getIntero() {
        return intero;
    }
}
```

Soluzione 15.s)

Tutte le affermazioni sono corrette tranne la numero 4. Infatti `AtomicInteger` è una classe non un'interfaccia.

Soluzione 15.t)

La compilazione dello snippet causerà tre errori, uno per ogni riga:

```
Esercizio15T.java:6: error: <anonymous Esercizio15T$1> is not abstract
  and does not override abstract method call() in Callable
        Callable<String> callable = new Callable<>() {public void
  call(){}};
```

```
                                                                ^
Esercizio15T.java:6: error: call() in <anonymous Esercizio15T$1>
  cannot implement call() in Callable
        Callable<String> callable = new Callable<>() {public void
  call(){}};
  ^
  return type void is not compatible with String
  where V is a type-variable:
    V extends Object declared in interface Callable
Esercizio15T.java:7: error: cannot find symbol
        Future<String> future =
  Executors.newFixedThreadPool(3).start(callable);
                                  ^
  symbol:   method start(Callable<String>)
  location: interface ExecutorService
3 errors
```

Infatti nella prima riga abbiamo provato a creare una classe anonima che estende `Callable` parametrizzata con una stringa. Poi viene invocato il metodo `call()` che ritorna `void` in luogo di `String`, che però non esiste. Nella seconda riga l'errore consiste nel chiamare il metodo `start()` (come per `Runnable`), ma il metodo da chiamare è `submit()`. Infine nella terza riga, viene invocato il metodo `get()` di `Future` che richiede la gestione dell'`InterruptedException`.
Quindi, il codice corretto sarebbe simile al seguente:

```
Callable<String> callable = new Callable<>() {
    public String call(){return "";}
};
ExecutorService service = Executors.newFixedThreadPool(3);
Future<String> future = service.submit(callable);
String result = null;
try {
    result = future.get();
}
catch (Exception exc) {
    exc.printStackTrace();
}
```

Soluzione 15.u)

Una possibile (e semplice) soluzione potrebbe essere la seguente:

```
import java.util.*;
import java.time.*;
import java.text.*;

public class Esercizio15U extends TimerTask {
```

```
    private Timer timer;

    public Esercizio15U() {
        timer = new Timer();
    }

    @Override
    public void run() {
        DateFormat timeFormatter =
          DateFormat.getTimeInstance(DateFormat.DEFAULT, Locale.getDefault());
        System.out.println("Sveglia! Sono le " + LocalTime.now());
        timer.cancel();
    }

    public static void main(String args[]) throws Exception {
        int seconds = Integer.parseInt(args[0])*1000;
        Esercizio15U timerTask = new Esercizio15U();
        timerTask.timer.schedule(timerTask, seconds);
    }
}
```

Soluzione 15.v)

Le risposte corrette sono la numero 2 e la numero 4.
La numero 1 definisce uno statement non compilabile, perché bisogna specificare quantomeno il numero di permits per quanto riguarda il costruttore di `Semaphore`.
La numero 3 è falsa perché `Semaphore` è una classe (facile intuirlo visto che viene istanziata con un costruttore).

Soluzione 15.z)

Le risposte corrette sono le numero 1 e 2.
La numero 3 è falsa perché `signalAll()` non è un metodo di `CyclicBarrier` ma di `Condition`, a cui abbiamo accennato tra le ultime righe del paragrafo 15.6.3.1.
La numero 4 è falsa, perché `CyclicBarrier` fornisce solo la possibilità ad un gruppo di thread di sincronizzarsi e attendersi l'un l'altro in un determinato punto del codice

Esercizi del capitolo 16

Espressioni Lambda

Le espressioni lambda e i reference a metodi non rappresentano un argomento semplice, ma sono indubbiamente una soluzione molto utile a certi problemi di programmazione. I seguenti esercizi dovrebbero permettere al lettore di comprendere meglio gli argomenti trattati nel capitolo 16. Anche in questo capitolo sono stati introdotti diversi esercizi che supportano la certificazione Oracle.

Esercizio 16.a) Espressioni lambda e reference a metodi:

1. Con un'espressione lambda è possibile fare tutto quello che fa una classe anonima.
2. Un'espressione lambda può utilizzare in maniera thread-safe le variabili d'istanza della classe in cui è dichiarata.
3. Un'espressione lambda può utilizzare le variabili d'istanza della classe in cui è definita solo se dichiarate `final`.
4. La seguente espressione lambda è legale:

```
Consumer <String> c = ((x)->System.out.println(x));
```

5. La seguente espressione lambda è legale:

```
(x, y) -> System.out.println(x);
          System.out.println(y);
```

6. Il seguente reference a metodo è legale:

```
System.out::println()
```

7. Il seguente statement è legale:

```
System.out.println(Math::random)
```

8. Il seguente statement è legale:

```
System.out::println(Math::random)
```

9. Il seguente statement è legale:

```
Supplier<Chitarra> chitarraSupplier = Chitarra::new;
```

10. Supponendo che la classe `Chitarra` abbia un costruttore senza parametri, allora il seguente codice è legale:

```
Chitarrista chitarrista = new Chitarrista();
chitarrista.suonaChitarra(Chitarra::new);
```

Esercizio 16.b)

Creare una classe `TestComparators` contenente un metodo `main()` che dichiara un array di stringhe (di cui almeno due con la stessa lunghezza). Creare degli oggetti `Comparator` con espressioni lambda o reference a metodi, per produrre ordinamenti secondo i seguenti criteri:

- in ordine di lunghezza (dalla stringa più lunga alla più breve)
- in ordine di lunghezza al contrario (dalla stringa più breve alla più lunga)
- in ordine alfabetico (usare comunque un'espressione lambda o un reference a metodo, anche se non ce ne sarebbe bisogno)
- in ordine alfabetico inverso
- in ordine di lunghezza e in caso di stessa lunghezza in ordine alfabetico

Ordinare l'array mediante il metodo `sort()` della classe `Arrays`, che prende in input come primo parametro l'array dichiarato e come secondo parametro un oggetto di tipo `Comparator`. Dopo ogni ordinamento stampare il risultato. Creare questa classe in un package come per esempio `com.claudiodesio.lambda.test`.

Esercizio 16.c)

Creare una classe `Citta` (non è possibile utilizzare le lettere accentate negli identificatori in Java), che astrae il concetto di città, che dichiara come variabili incapsulate la stringa `nome`, e i booleani `capoluogo`, e `diMare`.

> **Si ricorda che i metodi getter per i booleani solitamente usano come prefisso "is" in luogo di "get". Quindi i metodi `getCapoluogo()` e `getDiMare()`, si dovrebbero scrivere come `isDiMare()` e `isCapoluogo()`.**

Creare anche eventuali metodi di utilità come `toString()` (affinché ritorni solo il nome) e un costruttore. Questa classe apparterrà ad un package come per esempio `com.claudiodesio.lambda.dati`.
Creare una classe `Esercizio16C` con un metodo `main()`, che stampi:

- la lista di città di mare (per esempio `Città di mare: [Siracusa, Napoli, Pescara, Taranto]`)
- la lista di città capoluogo (per esempio `Città capoluogo: [Milano, Potenza, Perugia, Napoli]`)

sfruttando i metodi a reference e l'interfaccia `Predicate`.
Questa classe apparterrà ad un package come per esempio `com.claudiodesio.lambda.test`.

Esercizio 16.d)

Dopo aver svolto l'esercizio precedente, creare una classe `Esercizio16D` alternativa a `Esercizio16C`, che esegue le stesse operazioni usando le espressioni lambda.

Esercizio 16.e)

Creare la classe `Esercizio16E` modificando la classe `Esercizio16C` aggiungendo un metodo `stampaDettagli()` che stampi la lista delle città, anche con le informazioni definite dalle variabili `capoluogo` e `diMare`. Per esempio l'output potrebbe essere simile al seguente:

```
Milano è capoluogo,
Pescara è città di mare,
```

```
Napoli è capoluogo, è città di mare,
...
```

Per ottenere tale risultato, non bisogna cambiare la classe `Citta`, ma usare un'espressione lambda in `Esercizio16E`, sfruttando una delle interfacce funzionali standard, quale?

Esercizio 16.f)

Ripetere l'esercizio 10.h, utilizzando una espressione lambda in luogo della classe anonima. Riportiamo il testo dell'esercizio 10.h per comodità qui di seguito:

```
public class Persona {

    private String nome;
    private String cognome;
    private String dataDiNascita;
    private String professione;
    private String indirizzo;

    public Persona(String nome, String cognome) {
        this.nome = nome;
        this.cognome = cognome;
    }

    public Persona(String nome, String cognome, String dataDiNascita,
            String professione, String indirizzo) {
        this.nome = nome;
        this.cognome = cognome;
        this.dataDiNascita = dataDiNascita;
        this.professione = professione;
        this.indirizzo = indirizzo;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getCognome() {
        return cognome;
    }

    public void setCognome(String cognome) {
        this.cognome = cognome;
    }
```

```
    public String getDataDiNascita() {
        return dataDiNascita;
    }

    public void setDataDiNascita(String dataDiNascita) {
        this.dataDiNascita = dataDiNascita;
    }

    public String getProfessione() {
        return professione;
    }

    public void setProfessione(String professione) {
        this.professione = professione;
    }

    public String getIndirizzo() {
        return indirizzo;
    }

    public void setIndirizzo(String indirizzo) {
        this.indirizzo = indirizzo;
    }

    @Override
    public String toString() {
        return "Persona{nome=" + nome + ", cognome=" + cognome + "}";
    }
}
```

Supponiamo di voler scrivere un programma e di voler risfruttare la seguente classe `Persona`, ereditata da un programma già scritto e non modificabile. Purtroppo nel nostro contesto, avremmo bisogno di ridefinire il metodo `toString()` in modo che stampi non solo le informazioni sul nome e il cognome della persona, ma anche la data di nascita, l'indirizzo e la professione. Come già detto però, la classe è già in uso e non è possibile modificarla. In particolare il nostro requisito è che il metodo `toString()` restituisca la seguente stringa:

```
Nome:                   Arjen Anthony
Cognome:                Lucassen
Professione:            Compositore
Data di Nascita:   03/04/1960
Indirizzo:         Olanda
```

Creare quindi una classe `TestPersona` che ridefinisca in qualche modo il metodo `toString()` della classe `Persona` e stampi l'output di cui sopra.

Esercizio 16.g)

Consideriamo la classe `Osservazione` della soluzione dell'esercizio 15.c, che riportiamo di seguito per comodità:

```
package com.claudiodesio.osservatorio.test;

import com.claudiodesio.osservatorio.dati.Partecipante;
import com.claudiodesio.osservatorio.dati.Telescopio;

public class Osservazione {

    public static void main(String args[]) {
        Telescopio telescopio = new Telescopio();
        Partecipante[] partecipanti = getPartecipanti(telescopio);
        for (Partecipante partecipante : partecipanti) {
            partecipante.start();
        }
    }

    private static Partecipante[] getPartecipanti(Telescopio telescopio) {
        Partecipante[] partecipanti = {
            new Partecipante("Ciro", telescopio),
            new Partecipante("Gianluca", telescopio),
            new Partecipante("Pierluigi", telescopio),
            new Partecipante("Gigi", telescopio),
            new Partecipante("Nicola", telescopio),
            new Partecipante("Pino", telescopio),
            new Partecipante("Maurizio", telescopio),
            new Partecipante("Raffaele", telescopio),
            new Partecipante("Fabio", telescopio),
            new Partecipante("Vincenzo", telescopio)};
        return partecipanti;
    }
}
```

È possibile usare un'espressione lambda per implementare un override del metodo `run()` della classe `Partecipante`, per uno dei partecipanti?

Esercizio 16.h)

Considerato il seguente codice:

```
new Thread(()->System.out.print("Java");System.out.print("Java");).start();
```

quali delle seguenti affermazioni sono corrette?

1. Eseguendo questo snippet con JShell viene stampata la stringa `JavaJava`.
2. Eseguendo questo snippet con JShell viene stampata la stringa `Java`.

3. Questo snippet non supererà la compilazione.
4. Questo snippet non lancerà un'eccezione al runtime.
5. Questo snippet produrrà un warning in compilazione.

Esercizio 16.i)

Considerato il seguente codice:

```
new Thread((int a)->{
  int b = 0;
  b = a/b;
}).start();
```

quali delle seguenti affermazioni sono corrette?

1. Il codice non compila perché bisogna gestire una `ArithmeticException`.
2. Il codice compila e viene eseguito correttamente.
3. Il codice compila ma lancerà una `ArithmeticException` durante l'esecuzione.
4. Il codice produrrà un warning in compilazione.

Esercizio 16.l)

Supponiamo di avere un'espressione lambda che usa un metodo che lancia una checked exception senza gestirla. Quali delle seguenti affermazioni è corretta?

1. Il codice non compila perché bisogna gestire una checked exception nel blocco di codice dell'espressione lambda.
2. Il codice non compila, ma se è possibile, possiamo renderlo compilabile ridefinendo il metodo SAM dell'interfaccia funzionale gestendo l'eccezione all'interno del metodo stesso.
3. Il codice compila tranquillamente.
4. Dal momento che dobbiamo gestire l'eccezione, perdiamo le capacità di deduzione del compilatore, che ci porta a scrivere meno codice.

Esercizio 16.m)

Creare l'interfaccia funzionale che possa soddisfare il seguente uso di espressioni lambda:

```
Operazione operazione1 = (double a, double b) ->  a + b;
Operazione operazione2 = (double a, double b) ->  a - b;
Operazione operazione3 = (double a, double b) ->  a / b;
Operazione operazione4 = (double a, double b) ->  a * b;
```

Esercizio 16.n)

Esiste un'interfaccia funzionale standard, capace di sostituire l'interfaccia funzionale `Operazione` dell'esercizio precedente?

Usare la documentazione per risolvere l'esercizio.

Esercizio 16.o)

Riscrivere l'esercizio 16.b, sostituendo le espressioni lambda con reference a metodi.

Esercizio 16.p)

Come si chiama l'interfaccia funzionale che si adatta meglio a fungere da `Factory`?

1. `Predicate`
2. `Factory`
3. `Function`
4. `Supplier`
5. `Consumer`

Esercizio 16.q)

Selezionare le affermazioni corrette.
Un'interfaccia funzionale:

1. Deve essere annotata con l'annotazione `FunctionalInterface`.
2. Deve dichiarare un unico metodo.
3. Deve essere per forza implementata.
4. Non può essere estesa da un'altra interfaccia funzionale.
5. Non può essere estesa da un'altra interfaccia.

Esercizio 16.r)

Quali delle seguenti affermazioni sono corrette riguardo i reference a costruttore?

1. La sintassi è `NomeClasse::New`.
2. La sintassi è `NomeClasse::new()`.
3. La sintassi è `NomeClasse::new`.
4. Un reference a un costruttore si può assegnare ad un reference di un'interfaccia funzionale, il cui metodo SAM ritorna `void`.
5. Con un reference ad un costruttore, possiamo sostituire l'implementazione di un'interfaccia funzionale.

Esercizio 16.s)

Creare una classe `Persona` che dichiari le variabili `nome` e `anni` (nel senso di anni di età) e che implementi l'interfaccia `Comparable` in modo tale che il metodo ereditato `compareTo()` ordini per età crescente, e, nel caso di persone della stessa età, in ordine alfabetico (considerando il nome). Si faccia override anche del metodo `toString()`.
Poi, considerata la seguente classe:

```
import java.util.Arrays;

public class Esercizio16S {
    public static void main(String args[]) {
        Persona [] persone = {
            new Persona("Antonio",21),
            new Persona("Bruno",20),
            new Persona("Giorgio",19),
            new Persona("Martino",22),
            new Persona("Daniele",21)
        };
        Arrays.sort(persone, /*INSERISCI IL CODICE QUI*/);
        System.out.println(Arrays.toString(persone));
    }
}
```

si inserisca il codice corretto al posto del commento `/*INSERISCI IL CODICE QUI*/`, in modo tale da generare il seguente output:

```
[Giorgio, Bruno, Antonio, Daniele, Martino]
```

Esercizio 16.t)

Quali delle seguenti affermazioni sono corrette?

1. La sintassi è `NomeClasse::New`.
2. La sintassi è `NomeClasse::new()`.
3. La sintassi è `NomeClasse::new`.
4. Un reference a un costruttore si può assegnare ad un reference di un'interfaccia funzionale, il cui metodo SAM ritorna `void`.
5. Con un reference ad un costruttore, possiamo sostituire l'implementazione di un'interfaccia funzionale.

Esercizio 16.u)

Partendo dalla classe `Persona` dell'esercizio 16.s, e considerando la seguente classe:

```
import java.util.Arrays;
import java.util.function.BiPredicate;

public class Esercizio16U {
    public static void main(String args[]) {
        Persona [] persone = {
          new Persona("Antonio",21),
          new Persona("Bruno",20),
          new Persona("Giorgio",19),
          new Persona("Martino",22),
          new Persona("Daniele",21)
        };
        Persona personaCheIniziaPerD = getPersonaCheIniziaPer("D",
          persone, /*INSERISCI CODICE QUI*/);
        System.out.println(personaCheIniziaPerD);
    }

    static Persona getPersonaCheIniziaPer(String iniziale,
      Persona[] persone, BiPredicate<String, Persona> biPredicate) {
        for(Persona persona : persone) {
            if (biPredicate.test(iniziale, persona)) {
                return persona;
            }
        }
        return null;
    }
}
```

Quale espressione lambda è possibile inserire al posto del commento `/*INSERISCI CODICE QUI*/` per recuperare il primo oggetto `Persona` dell'array `persone` che ha un nome che inizia con la "D"?

Esercizio 16.v)

Quali delle seguenti affermazioni sono corrette?

1. Con le espressioni lambda il reference `this` si riferisce direttamente alla classe in cui è inclusa l'espressione.
2. Per le espressioni lambda valgono le stesse regole che valgono per le classi anonime.
3. La sintassi di un'espressione lambda permette di utilizzare altre espressioni lambda innestate.
4. La sintassi di un'espressione lambda permette di utilizzare reference a metodi innestati.
5. La sintassi di un reference a metodo permette di utilizzare altre espressioni lambda innestate.
6. La sintassi di un reference a metodo permette di utilizzare reference a metodi innestati.

Esercizio 16.z)

Quali delle seguenti affermazioni sono corrette?

1. L'interfaccia funzionale che dichiara due tipi generici come parametri in input e ne ritorna un altro, si chiama `BiFunction`.
2. L'interfaccia funzionale che non dichiara tipi generici come parametri in input, ma ne ritorna un altro, si chiama `Provider`.
3. L'interfaccia funzionale che non dichiara tipi generici come parametri in input, ma ne ritorna un altro, si chiama `Supplier`.
4. L'interfaccia funzionale che dichiara un tipo generico come parametro in input e ne ritorna un altro, si chiama `Function`.
5. L'interfaccia funzionale che dichiara tre tipi generici come parametri in input e ne ritorna un altro, si chiama `TriFunction`.
6. L'interfaccia funzionale che dichiara un tipo generico come parametro in input e ritorna lo stesso tipo, si chiama `UnaryOperator`.
7. È possibile concatenare più `UnaryOperator` tramite il metodo di default `and()`.
8. L'interfaccia funzionale che dichiara un tipo generico come parametro in input e non ritorna nulla, si chiama `Consumer`.

Soluzioni degli esercizi del capitolo 16

Soluzione 16.a) Espressioni lambda e reference a metodi:

1. **Falso**, per esempio in una classe anonima possiamo definire anche variabili d'istanza.
2. **Falso**.
3. **Falso**, cfr. paragrafo 16.1.3.2
4. **Falso**.
5. **Falso**, mancano le parentesi graffe che circondano le istruzioni del metodo.
6. **Falso**, le parentesi vicino al metodo `println` non fanno parte della sintassi.
7. **Falso**.
8. **Falso**.
9. **Vero**.
10. **Vero**.

Soluzione 16.b)

Il listato richiesto potrebbe essere come il seguente:

```
package com.claudiodesio.lambda.test;
```

```
import java.util.Arrays;
import java.util.Comparator;

public class TestComparators {

    static String nomi[] = {"Clarissa", "Jem", "Top", "Ermeringildo",
      "Iamaca", "Tom", "Arlequin", "Francesca", "Cumbus", "Blue"};

    public static void main(String args[]) {

        Comparator<String> comparatorLunghezza = (first, second)
                -> -(Integer.compare(first.length(), second.length()));

        Comparator<String> comparatorLunghezzaAlContrario = (first, second)
                -> (Integer.compare(first.length(), second.length()));

        Comparator<String> comparatorAlfabeticoAlContrario = (first, second)
                -> -(first.compareTo(second));

        Comparator<String> comparatorLunghezzaEAlfabeticoAlContrario =
          (first, second)
                -> {
                    int result = -Integer.compare(first.length(),
                        second.length());
                    if (result == 0) {
                        result = first.compareTo(second);
                    }
                    return result;
                };
        Arrays.sort(nomi, comparatorLunghezza);
        System.out.println("Nomi ordinati per lunghezza: " +
          Arrays.asList(nomi));

        Arrays.sort(nomi, comparatorLunghezzaAlContrario);
        System.out.println("Nomi ordinati per lunghezza al contrario: " +
          Arrays.asList(nomi));

        Arrays.sort(nomi, String::compareTo);
        System.out.println("Nomi ordinati: " + Arrays.asList(nomi));

        Arrays.sort(nomi, comparatorAlfabeticoAlContrario);
        System.out.println("Nomi ordinati al contrario: " +
          Arrays.asList(nomi));

        Arrays.sort(nomi, comparatorLunghezzaEAlfabeticoAlContrario);
        System.out.println(
          "Nomi ordinati per lunghezza al contrario e in ordine alfabetico: "
          + Arrays.asList(nomi));
    }
}
```

Si noti che il metodo `asList()` della classe `Arrays` è stato usato solo per sfruttare la rappresentazione testuale della collezione.
L'output sarà:

```
Nomi ordinati per lunghezza: [Ermeringildo, Francesca, Clarissa,
 Arlequin, Iamaca, Cumbus, Blue, Jem, Top, Tom]
Nomi ordinati per lunghezza al contrario: [Jem, Top, Tom, Blue, Iamaca,
 Cumbus, Clarissa, Arlequin, Francesca, Ermeringildo]
Nomi ordinati: [Arlequin, Blue, Clarissa, Cumbus, Ermeringildo,
 Francesca, Iamaca, Jem, Tom, Top]
Nomi ordinati al contrario: [Top, Tom, Jem, Iamaca, Francesca,
 Ermeringildo, Cumbus, Clarissa, Blue, Arlequin]
Nomi ordinati per lunghezza al contrario e in ordine alfabetico:
 [Ermeringildo, Francesca, Arlequin, Clarissa, Cumbus, Iamaca, Blue,
 Jem, Tom, Top]
```

Soluzione 16.c)

Il listato della classe `Citta`, dovrebbe essere il seguente:

```
package com.claudiodesio.lambda.dati;

public class Citta {

    private String nome;

    private boolean capoluogo;

    private boolean diMare;

    public Citta(String nome, boolean capoluogo, boolean diMare) {
        this.nome = nome;
        this.capoluogo = capoluogo;
        this.diMare = diMare;
    }

    public boolean isDiMare() {
        return diMare;
    }

    public void setDiMare(boolean diMare) {
        this.diMare = diMare;
    }

    public boolean isCapoluogo() {
        return capoluogo;
    }

    public void setCapoluogo(boolean capoluogo) {
```

```
        this.capoluogo = capoluogo;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    @Override
    public String toString() {
        return getNome();
    }
}
```

Il listato della classe `Esercizio16C` invece, potrebbe essere codificato nel seguente modo:

```
package com.claudiodesio.lambda.test;

import com.claudiodesio.lambda.dati.Citta;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Predicate;

public class Esercizio16C {

    public static void main(String args[]) {
        List<Citta> listaCitta = getCitta();
        System.out.println("Città di mare: " +
          filtraCitta(listaCitta, Esercizio16C::isDiMare));
        listaCitta = getCitta();
        System.out.println("Città capoluogo: " +
          filtraCitta(listaCitta, Esercizio16C::isCapoluogo));
        listaCitta = getCitta();
    }

    public static List<Citta> filtraCitta(List<Citta> listaCitta,
                                          Predicate<Citta> p) {
        final Iterator<Citta> iterator = listaCitta.iterator();
        while (iterator.hasNext()) {
            Citta citta = iterator.next();
            if (!p.test(citta)) {
                iterator.remove();
            }
        }
        return listaCitta;
```

```
    }

    private static List<Citta> getCitta() {
        List<Citta> citta = new ArrayList<>();
        citta.add(new Citta("Milano", true, false));
        citta.add(new Citta("Rovigo", false, false));
        citta.add(new Citta("Potenza", true, false));
        citta.add(new Citta("Siracusa", false, true));
        citta.add(new Citta("Perugia", true, false));
        citta.add(new Citta("Napoli", true, true));
        citta.add(new Citta("Pescara", false, true));
        citta.add(new Citta("Taranto", false, true));
        citta.add(new Citta("Siena", false, false));
        return citta;
    }

    public static boolean isDiMare(Citta citta) {
        return citta.isDiMare();
    }

    public static boolean isCapoluogo(Citta citta) {
        return citta.isCapoluogo();
    }
}
```

L'output sarà:

```
Città di mare: [Siracusa, Napoli, Pescara, Taranto]
Città capoluogo: [Milano, Potenza, Perugia, Napoli]
```

Soluzione 16.d)

Il listato della classe `Esercizio16D` potrebbe essere il seguente:

```
package com.claudiodesio.lambda.test;

import com.claudiodesio.lambda.dati.Citta;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.function.Predicate;

public class Esercizio16D {

    public static void main(String args[]) {
        List<Citta> listCitta = getCitta();
        System.out.println("Città di mare: " +
          filtraCitta(listCitta, (citta) -> citta.isDiMare()));
        listCitta = getCitta();
        System.out.println("Città capoluogo: " +
```

```
            filtraCitta(listCitta, (citta) -> citta.isCapoluogo()));
    }

    public static List<Citta> filtraCitta(List<Citta> listaCitta,
                                          Predicate<Citta> p) {
        final Iterator<Citta> iterator = listaCitta.iterator();
        while (iterator.hasNext()) {
            Citta citta = iterator.next();
            if (!p.test(citta)) {
                iterator.remove();
            }
        }
        return listaCitta;
    }

    private static List<Citta> getCitta() {
        List<Citta> citta = new ArrayList<>();
        citta.add(new Citta("Milano", true, false));
        citta.add(new Citta("Rovigo", false, false));
        citta.add(new Citta("Potenza", true, false));
        citta.add(new Citta("Siracusa", false, true));
        citta.add(new Citta("Perugia", true, false));
        citta.add(new Citta("Napoli", true, true));
        citta.add(new Citta("Pescara", false, true));
        citta.add(new Citta("Taranto", false, true));
        citta.add(new Citta("Siena", false, false));
        return citta;
    }
}
```

L'output rimarrà lo stesso dell'esercizio precedente:

```
Città di mare: [Siracusa, Napoli, Pescara, Taranto]
Città capoluogo: [Milano, Potenza, Perugia, Napoli]
```

Soluzione 16.e)

La classe `Esercizio16E` potrebbe essere la seguente:

```
package com.claudiodesio.lambda.test;

import com.claudiodesio.lambda.dati.Citta;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Predicate;

public class Esercizio16E {
```

```
public static void main(String args[]) {
    List<Citta> listaCitta = getCitta();
    System.out.println("Città di mare: " +
      filtraCitta(listaCitta, Esercizio16E::isDiMare));
    listaCitta = getCitta();
    System.out.println("Città capoluogo: " +
      filtraCitta(listaCitta, Esercizio16E::isCapoluogo));
    listaCitta = getCitta();
    stampaDettagli(listaCitta, (citta) -> System.out.println(citta.getNome()
      + (citta.isCapoluogo() ? " è capoluogo," : "")
      + (citta.isDiMare() ? " è città di mare," : "")));
}

public static List<Citta> filtraCitta(List<Citta> listaCitta,
                                       Predicate<Citta> p) {
    final Iterator<Citta> iterator = listaCitta.iterator();
    while (iterator.hasNext()) {
        Citta citta = iterator.next();
        if (!p.test(citta)) {
            iterator.remove();
        }
    }
    return listaCitta;
}

public static void stampaDettagli(List<Citta> listaCitta,
                                  Consumer<Citta> p) {
    for (Citta citta : listaCitta) {
        p.accept(citta);
    }
}

private static List<Citta> getCitta() {
    List<Citta> citta = new ArrayList<>();
    citta.add(new Citta("Milano", true, false));
    citta.add(new Citta("Rovigo", false, false));
    citta.add(new Citta("Potenza", true, false));
    citta.add(new Citta("Siracusa", false, true));
    citta.add(new Citta("Perugia", true, false));
    citta.add(new Citta("Napoli", true, true));
    citta.add(new Citta("Pescara", false, true));
    citta.add(new Citta("Taranto", false, true));
    citta.add(new Citta("Siena", false, false));
    return citta;
}

public static boolean isDiMare(Citta citta) {
    return citta.isDiMare();
}
```

```
    public static boolean isCapoluogo(Citta citta) {
        return citta.isCapoluogo();
    }
}
```

E quindi l'interfaccia funzionale da usare era l'interfaccia `Consumer`.
È bastato poi invocare questo metodo usando un'espressione lambda (ed un paio di operatori ternari) nel metodo `main()`:

```
stampaDettagli(listaCitta, (citta) -> System.out.println(citta.getNome()
  + (citta.isCapoluogo() ? " è capoluogo, " : "")
  + (citta.isDiMare() ? " è città di mare" : "")));
```

L'output sarà:

```
Città di mare: [Siracusa, Napoli, Pescara, Taranto]
Città capoluogo: [Milano, Potenza, Perugia, Napoli]
Milano è capoluogo,
Rovigo
Potenza è capoluogo,
Siracusa è città di mare,
Perugia è capoluogo,
Napoli è capoluogo, è città di mare,
Pescara è città di mare,
Taranto è città di mare,
Siena
```

Soluzione 16.f)

È impossibile in questo caso sostituire la classe anonima con un'espressione lambda! Ricordiamo che un'espressione lambda funziona implementando interfacce funzionali (con un solo metodo astratto).

Soluzione 16.g)

Anche in questo caso, è impossibile usare un'espressione lambda. Invece è possibile utilizzare una classe anonima:

```
package com.claudiodesio.osservatorio.test;

import com.claudiodesio.osservatorio.dati.Partecipante;
import com.claudiodesio.osservatorio.dati.Telescopio;

public class Osservazione {

    public static void main(String args[]) {
        Telescopio telescopio = new Telescopio();
        Partecipante[] partecipanti = getPartecipanti(telescopio);
```

```
        for (Partecipante partecipante : partecipanti) {
            partecipante.start();
        }
    }

    private static Partecipante[] getPartecipanti(Telescopio telescopio) {
        Partecipante[] partecipanti = {
            new Partecipante("Ciro", telescopio),
            new Partecipante("Gianluca", telescopio),
            new Partecipante("Pierluigi", telescopio),
            new Partecipante("Gigi", telescopio),
            new Partecipante("Nicola", telescopio) {
                @Override
                public void run() {
                    System.out.println(getNome() + " sono pronto!");
                    super.run();
                }

            },
            new Partecipante("Pino", telescopio),
            new Partecipante("Maurizio", telescopio),
            new Partecipante("Raffaele", telescopio),
            new Partecipante("Fabio", telescopio),
            new Partecipante("Vincenzo", telescopio)};
        return partecipanti;
    }
}
```

Soluzione 16.h)

Solo la terza affermazione è corretta. L'output di JShell è il seguente:

```
|  Error:
|  ')' expected
|  new Thread(()->System.out.print("Java");System.out.print("Java");).start();
|
```

Infatti mancano le parentesi graffe intorno al blocco di codice. Se ci fossero:

```
new Thread(()->{System.out.print("Java");System.out.print("Java");}).start();
```

allora la prima affermazione sarebbe stata quella corretta:

```
jshell> JavaJava
```

Soluzione 16.i)

Nessuna delle affermazioni è corretta. Infatti il codice non compilerà perché al costruttore di un `Thread` bisogna passare un'implementazione del metodo `run()`,

che non prende in input parametri come specificato nel codice. Eseguendo questo snippet su JShell otterremo il seguente output:

```
jshell> new Thread((int a)->{
   ...>   int b = 0;
   ...>   b = a/b;
   ...> }).start();
|  Error:
|  no suitable constructor found for Thread((int a)->{[...] b; })
|   constructor java.lang.Thread.Thread(java.lang.Runnable) is not applicable
|   (argument mismatch; incompatible parameter types in lambda expression)
|   constructor java.lang.Thread.Thread(java.lang.String) is not applicable
|   (argument mismatch; java.lang.String is not a functional interface)
|  new Thread((int a)->{
|  ^--------------------...
```

Notiamo che `ArithmeticException` estende `RuntimeException` non una checked exception. Il compilatore quindi non avrebbe segnalato errori, e per questa ragione la prima affermazione era sicuramente scorretta. La seconda e la quarta affermazione sono ovviamente false. La terza affermazione sarebbe stata corretta se l'implementazione del metodo `run()` fosse stata corretta, ma non lo è.

Soluzione 16.l)

Le affermazioni sono tutte corrette tranne la numero 3.

Soluzione 16.m)

La soluzione è banale:

```
public interface Operazione {
    double operazione(double x, double y);
}
```

Soluzione 16.n)

Sì, esiste la classe `DoubleBinaryOperator` che definisce il metodo:

```
double applyAsDouble(double left, double right)
```

che coincide come parametri e tipo di ritorno con il metodo `operazione()` dell'interfaccia `Operazione` dell'esercizio precedente (ovviamente il nome non conta).

Soluzione 16.o)

La soluzione potrebbe essere la seguente:

```
package com.claudiodesio.lambda.test;
```

```
import java.util.Arrays;
import java.util.Comparator;

public class Esercizio16O {

    static String nomi[] = {"Clarissa","Jem","Top","Ermeringildo",
      "Iamaca","Tom","Arlequin","Francesca","Cumbus","Blue"
    } ;

    static int compareLunghezza(String first, String second) {
        return -(Integer.compare(first.length(), second.length()));
    }

    static int compareLunghezzaAlContrario(String first, String second) {
        return (Integer.compare(first.length(), second.length()));
    }

    static int compareAlfabeticoAlContrario(String first, String second) {
        return -(first.compareTo(second));
    }

    static int compareLunghezzaEAlfabeticoAlContrario(String first,
                                                      String second) {
        int result = -Integer.compare(first.length(),
        second.length());
        if (result == 0) {
            result = first.compareTo(second);
        }
        return result;
    }

    public static void main(String args[]) {

        Arrays.sort(nomi, Esercizio16O::compareLunghezza);
        System.out.println("Nomi ordinati per lunghezza: "
          + Arrays.asList(nomi));

        Arrays.sort(nomi, Esercizio16O::compareLunghezzaAlContrario);
        System.out.println("Nomi ordinati per lunghezza al contrario: "
          + Arrays.asList(nomi));

        Arrays.sort(nomi, String::compareTo);
        System.out.println("Nomi ordinati : " + Arrays.asList(nomi));

        Arrays.sort(nomi, Esercizio16O::compareAlfabeticoAlContrario);
        System.out.println("Nomi ordinati al contrario: "
          + Arrays.asList(nomi));

        Arrays.sort(nomi,
          Esercizio16O::compareLunghezzaEAlfabeticoAlContrario);
```

```
        System.out.println(
          "Nomi ordinati per lunghezza al contrario e in ordine alfabetico: "
          + Arrays.asList(nomi));
    }
}
```

Soluzione 16.p)

L'esercizio era davvero semplice, la risposta giusta è la numero 4, `Supplier`, in quanto la definizione del suo metodo è la seguente:

```
T get()
```

dove `T` è un parametro generico dell'interfaccia.
Si noti che l'interfaccia funzionale `Factory` della risposta 2 non esiste.

Soluzione 16.q)

Nessuna delle affermazioni è corretta.
Solamente la numero 2 potrebbe non risultare scorretta in maniera evidente. Infatti un'interfaccia funzionale deve dichiarare un unico metodo astratto, ma è possibile dichiarare anche metodi di default e statici, sia pubblici che privati.

Soluzione 16.r)

Le affermazioni corrette sono la numero 3 (il che esclude la correttezza della 1 e della 2) e la numero 5 (che è coerente con la definizione delle espressioni lambda). L'affermazione numero 4 è scorretta perché il metodo SAM non può restituire `void`, ma deve restituire lo stesso tipo del costruttore.

Soluzione 16.s)

La classe `Persona` potrebbe essere la seguente:

```
import java.util.*;

public class Persona implements Comparable<Persona> {

    private String nome;

    private int anni;

    public Persona(String nome, int anni) {
        this.nome = nome;
        this.anni = anni;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
```

```
    public String getNome() {
        return nome;
    }

    public void setAnni(int anni) {
        this.anni = anni;
    }

    public int getAnni() {
        return anni;
    }

    @Override
    public int compareTo(Persona altraPersona) {
        int result = Integer.valueOf(this.anni).compareTo(
                     Integer.valueOf(altraPersona.anni));
        if (result == 0) {
            result = this.nome.compareTo(altraPersona.nome);
        }
        return result;
    }

    public String toString() {
        return nome;
    }
}
```

Invece la classe `Esercizio16S` può essere completata con il "reference a metodo d'istanza di un certo tipo" (che abbiamo studiato nel paragrafo 16.2.4) nel seguente modo (in grassetto il codice aggiunto):

```
import java.util.Arrays;

public class Esercizio16S {
    public static void main(String args[]) {
        Persona [] persone = {
            new Persona("Antonio",21),
            new Persona("Bruno",20),
            new Persona("Giorgio",19),
            new Persona("Martino",22),
            new Persona("Daniele",21)
        };
        Arrays.sort(persone, Persona::compareTo);
        System.out.println(Arrays.toString(persone));
    }
}
```

Soluzione 16.t)

La risposta giusta è la numero 6: `BiPredicate`. Infatti essa definisce il metodo:

```
boolean test(T t, U u)
```

Soluzione 16.u)

La soluzione potrebbe essere la seguente (il codice aggiunto è in grassetto):

```
import java.util.Arrays;
import java.util.function.BiPredicate;

public class Esercizio16U {
    public static void main(String args[]) {
        Persona [] persone = {
            new Persona("Antonio",21),
            new Persona("Bruno",20),
            new Persona("Giorgio",19),
            new Persona("Martino",22),
            new Persona("Daniele",21)
        };
        Persona personaCheIniziaPerD = getPersonaCheIniziaPer("D", persone,
          (String iniziale, Persona persona) ->
          persona.getNome().startsWith(iniziale));
        System.out.println(personaCheIniziaPerD);
    }

    static Persona getPersonaCheIniziaPer(String iniziale, Persona[] persone,
      BiPredicate<String, Persona> biPredicate){
        for(Persona persona : persone) {
            if (biPredicate.test(iniziale, persona)) {
                return persona;
            }
        }
        return null;
    }
}
```

Soluzione 16.v)

Le risposte corrette sono la numero 1, 3 e 4. La numero 2 è falsa, anzi il fatto che le espressioni lambda non ereditino le complicate regole delle classi anonime è uno dei vantaggi dell'utilizzare espressioni lambda al posto delle classi anonime. Le numero 5 e 6 sono false perché la sintassi di un reference a metodo non prevede codice innestato, e quindi è impossibile utilizzare altre espressioni lambda innestate o altri reference a metodo.

Soluzione 16.z)

Le risposte corrette sono la numero 1, 3, 4 e 7. La numero 2 è falsa visto che è corretta la numero 3.

La numero 5 è falsa perché `TriFunction` non esiste, ma si potrebbe creare facilmente. La numero 6 è falsa perché non esiste il metodo `and()` all'interno di `UnaryOperator` (semmai esiste `andThen()`).

Esercizi del capitolo 17

Collections Framework e Stream API

Le collection sono tra le implementazioni più utilizzate in assoluto in Java. La documentazione è fondamentale, per essere tra i migliori programmatori bisognerebbe consultarla assiduamente. Essendo una libreria molto estesa, ci sarà sempre un metodo o un'implementazione che fa al caso nostro, che non abbiamo mai utilizzato. La Stream API ora ha ulteriormente ampliato gli orizzonti d'applicazione delle collection. Consigliamo (come già fatto nell'introduzione) di commentare ogni singola riga implementata per memorizzare meglio le definizioni e il significato di alcuni statement.

Esercizio 17.a) Framework Collections, Vero o Falso:

1. `Collection` , `Map`, `SortedMap`, `Set`, `List` e `SortedSet` sono interfacce e non possono essere istanziate.
2. Un `Set` è una collezione ordinata di oggetti; una `List` non ammette elementi duplicati ed è ordinata.
3. Le mappe non possono contenere chiavi duplicate ed ogni chiave può essere associata ad un solo valore.
4. Esistono diverse implementazioni astratte da personalizzare nel framework come `AbstractMap`.

5. Una `HashMap` è più performante rispetto ad una `Hashtable` perché non è sincronizzata.
6. Una `HashMap` è più performante rispetto ad un `TreeMap` ma quest'ultima, essendo un'implementazione di `SortedMap`, gestisce l'ordinamento.
7. `HashSet` è più performante rispetto a `TreeSet` ma non gestisce l'ordinamento.
8. `Iterator` ed `Enumeration` hanno lo stesso ruolo ma quest'ultima permette durante le iterazioni di rimuovere anche elementi.
9. `ArrayList` ha prestazioni migliori rispetto a `Vector` perché non è sincronizzato, ma entrambi hanno meccanismi per ottimizzare le prestazioni.
10. La classe `Collections` è una lista di `Collection`.

Esercizio 17.b) Stream API, Vero o Falso:

1. `Stream` è una classe che implementa `Collection`.
2. È possibile iterare su uno stream in entrambe le direzioni.
3. Una pipeline è costituita da una sorgente, metodi di aggregazione opzionali e un metodo terminale.
4. Il metodo `map()` è da considerarsi un metodo di aggregazione.
5. `Optional` è un'interfaccia che permette di evitare di avere a che fare con le `NullPointerException`.
6. I metodi di riduzione sono operazioni di aggregazione.
7. Il metodo `joining()` di `Stream` permette di concatenare stringhe con dei separatori di tipo stringa.
8. La classe `DoubleSummaryStatistics` è un particolare stream.
9. Il metodo `parallelStream()` ritorna uno stream capace di usare l'algoritmo Fork/Join per eseguire operazioni sugli elementi di una collection.
10. Uno stream è istanziabile solo a partire da una collezione.

Esercizio 17.c)

Creare una collection che, nel caso si aggiungano due elementi uguali, lanci un'eccezione personalizzata (da creare anch'essa). Creare infine anche una classe di test.

Esercizio 17.d)

Creare una mappa chiamata `MappaIncrementale`, che permette di aggiungere più valori (ordinati in ordine di aggiunta) alla stessa chiave. Creare anche una classe di test.

Esercizio 17.e)

Sfruttando le classi create negli esercizi precedenti creare un'estensione di `MappaIncrementale` (chiamiamola `MappaIncrementaleRobusta`) che, quando si aggiunge un elemento la cui chiave esiste già, lancia un'eccezione come descritto nell'esercizio 17.c. Creare anche una classe di test.

Esercizio 17.f)

Dopo aver svolto l'esercizio precedente, si riempia la classe di test di oggetti `Citta`, la cui classe è stata creata nell'esercizio 16.c e che riportiamo di seguito per comodità:

```
package com.claudiodesio.dati;

public class Citta {

    private String nome;

    private boolean capoluogo;

    private boolean diMare;

    public Citta(String nome, boolean capoluogo, boolean diMare) {
        this.nome = nome;
        this.capoluogo = capoluogo;
        this.diMare = diMare;
    }

    public boolean isDiMare() {
        return diMare;
    }

    public void setDiMare(boolean diMare) {
        this.diMare = diMare;
    }

    public boolean isCapoluogo() {
        return capoluogo;
    }

    public void setCapoluogo(boolean capoluogo) {
```

```
        this.capoluogo = capoluogo;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    @Override
    public String toString() {
        return getNome();
    }
}
```

Aggiungere metodi `equals()` e `hashcode()`. Con uno stream stampare tutte le città di mare.
Con un secondo stream stampare tutte le città capoluogo. Con un terzo stream stampare tutte le città che finiscono con la lettera 'a'.

Esercizio 17.g) Riguardo il ciclo for migliorato, quali delle seguenti affermazioni sono corrette?

1. Il ciclo `for` migliorato può in ogni caso sostituire un ciclo `for`.
2. Il ciclo `for` migliorato può essere utilizzato con gli array e con le classi che implementano `Iterable`.
3. Il ciclo `for` migliorato sostituisce l'utilizzo di `Iterator`.
4. Il ciclo `for` migliorato non può sfruttare correttamente i metodi di `Iterator`.
5. In un ciclo `for` migliorato non è possibile effettuare cicli all'indietro.

Esercizio 17.h)

Considerato il seguente codice:

```
import java.util.*;

public class Esercizio17H {
    public static void main(String args[]) {
        Collection map = new HashMap(2);
        map.put(1,1);
        System.out.println(map);
```

```
        }
    }
```

Quale tra le seguenti affermazioni è corretta?

1. Il codice viene compilato con dei warning, ed eseguito stampa `{1=1}`.
2. Il codice viene compilato con dei warning, ma lancia un'eccezione durante l'esecuzione.
3. Il codice non compila.
4. Il codice viene compilato correttamente, ed eseguito stampa `{1=1}`.
5. Il codice viene compilato con dei warning, ed eseguito stampa `{1=1, null=null}`.
6. Il codice viene compilato con dei warning, ed eseguito stampa `{1=1, 0=0}`.

Esercizio 17.i)

Quali delle seguenti affermazioni sono corrette?

1. L'interfaccia `Iterable` dichiara il metodo `forEach`.
2. `Iterator` estende `Iterable`.
3. `Iterator` definisce il metodo `forEachRemaining()`.
4. `Collection` implementa l'interfaccia `Iterable`.

Esercizio 17.l)

Quali delle seguenti affermazioni sono corrette?

1. `Collection` è una superclasse di `List`.
2. Una `Collection` può essere trasformata in un array invocando il metodo `toArray()` definito nella classe `Arrays`.
3. Un array può essere trasformato in una `Collection` mediante il metodo `toCollection()` della classe `Arrays`.
4. `Collection` definisce il metodo `add()`.

Esercizio 17.m)

Consideriamo la seguente classe:

```
import java.util.*;

public class Esercizio17M {
    public static void main(String args[]) {
        ArrayList<String> list = new ArrayList<>(3);
        list.add("*");
        list.add("@");
        list.set(1, "$");
        ListIterator listIterator = list.listIterator();
        while(listIterator.hasNext()) {
            System.out.println(listIterator.next());
        }
        while(listIterator.hasPrevious()) {
            System.out.println(listIterator.previous());
        }
    }
}
```

Se eseguiamo questa classe, quale sarà l'output?

Esercizio 17.n)

Si crei un semplice programma che definisca un `ArrayList` di interi, e che venga riempito con i primi 50 numeri pari nella maniera più efficiente.

Esercizio 17.o)

Quali delle seguenti affermazioni sono corrette?

1. Un'implementazione di `Set` non può ordinare i suoi elementi.
2. Un'implementazione di `Set` non ammette più di un elemento `null`.
3. L'interfaccia `Set` è estesa da `SortedSet`.
4. L'implementazione `HashSet` non è thread safe.

Esercizio 17.p)

Quali delle seguenti affermazioni sono corrette?

1. La classe `Vector` è un'implementazione di `List`.
2. Un'implementazione di `List` non ammette elementi `null`.
3. Un reference di tipo `List` può diventare thread safe se gli viene assegnato un oggetto di tipo `List` che viene ritornato dal metodo `synchronizedList()`.
4. Per eliminare gli elementi duplicati da una lista, basta riempire un `Set` con gli elementi della `List`.

Esercizio 17.q)

Data la classe:

```
import java.util.*;

public class ClaudioLinkedList extends LinkedList<String> {
    public ClaudioLinkedList() {
      add("X");
      add("L");
      add("W");
      add("U");
      add("D");
      add("I");
      add("Z");
    }
}
```

aggiungere nella seguente classe `Esercizio17Q`:

```
import java.util.*;

public class Esercizio17Q {

   public static void main(String args[]) {

      ClaudioLinkedList claudioLinkedList = new ClaudioLinkedList();
      /*INSERISCI CODICE QUI*/
      System.out.println(claudioLinkedList);
   }
}
```

il codice al posto del commento `INSERISCI CODICE QUI`, che permetterà di generare il seguente output:

```
[C, L, A, U, D, I, O]
```

Esercizio 17.r)

Se vogliamo utilizzare una mappa in maniera thread-safe, che opzioni abbiamo? Elencare almeno due opzioni.

Esercizio 17.s)

Se vogliamo utilizzare una lista immutabile, che opzioni abbiamo? Elencare almeno due opzioni. E se vogliamo utilizzare un set immutabile, che opzioni abbiamo? Elencare almeno due opzioni.

Esercizio 17.t)

Considerata la lista di stringhe che ritorna il seguente metodo:

```
public static List<String> getStringList() {
    String stringa = "I ragazzi che si amano si baciano in piedi "
      + "Contro le porte della notte "
      + "E i passanti che passano li segnano a dito "
      + "Ma i ragazzi che si amano "
      + "Non ci sono per nessuno "
      + "Ed è la loro ombra soltanto "
      + "Che trema nella notte "
      + "Stimolando la rabbia dei passanti "
      + "La loro rabbia il loro disprezzo le risa la loro invidia "
      + "I ragazzi che si amano non ci sono per nessuno "
      + "Essi sono altrove molto più lontano della notte "
      + "Molto più in alto del giorno "
      + "Nell'abbagliante splendore del loro primo amore ";
    String[] stringhe = stringa.split(" ");
    return Arrays.asList(stringhe);
}
```

scrivere una pipeline che consideri solo le parole che non iniziano con "a", e ne calcoli (e stampi) la lunghezza media.

Esercizio 17.u)

Quali delle seguenti affermazioni sono corrette:

1. `Optional` è un'interfaccia generica.
2. Il metodo `ofNullable()` restituisce un oggetto `Optional` che fa da wrapper all'oggetto passato in input.
3. `orElseThrow()` è un metodo di `Optional` che restituisce un oggetto `Optional` o lancia un'eccezione che può essere specificata in input, nel caso l'oggetto "wrappato" sia `null`.
4. `findFirst()` è un metodo di `Optional` che restituisce un oggetto `Optional` o `null`, nel caso l'oggetto "wrappato" sia `null`.

Esercizio 17.v)

Considerato il seguente codice:

```
import java.util.*;
import java.util.stream.*;
```

```
public class Esercizio17V {

    public static void main(String args[]) {
        List<String> stringList = getStringList();
        /*INSERISCI CODICE QUI*/
        System.out.println(map);
    }

    public static List<String> getStringList() {
        String stringa = "I ragazzi che si amano si baciano in piedi "
          + "Contro le porte della notte "
          + "E i passanti che passano li segnano a dito "
          + "Ma i ragazzi che si amano "
          + "Non ci sono per nessuno "
          + "Ed è la loro ombra soltanto "
          + "Che trema nella notte "
          + "Stimolando la rabbia dei passanti "
          + "La loro rabbia il loro disprezzo le risa la loro invidia "
          + "I ragazzi che si amano non ci sono per nessuno "
          + "Essi sono altrove molto più lontano della notte "
          + "Molto più in alto del giorno "
          + "Nell'abbagliante splendore del loro primo amore ";
        String[] stringhe = stringa.split(" ");
        return Arrays.asList(stringhe);
    }
}
```

scrivere una pipeline per creare una mappa che raggruppi le parole del testo che iniziano per la stessa iniziale, ignorando il fatto che l'iniziale sia maiuscola o minuscola. Inserire tale pipeline al posto del commento `INSERISCI CODICE QUI`.

Esercizio 17.z)

Partendo dal risultato dell'esercizio 17.v, sostituire all'istruzione di stampa:

```
        System.out.println(map);
```

una semplice pipeline che stampi riga per riga il contenuto della mappa.

Soluzioni degli esercizi del capitolo 17

Soluzione 17.a) Framework Collections, Vero o Falso:

1. **Vero**.
2. **Falso**.
3. **Vero**.
4. **Vero**.
5. **Vero**.
6. **Vero**.
7. **Vero**.
8. **Falso**.
9. **Vero**.
10. **Falso**.

Soluzione 17.b) Stream API, Vero o Falso:

1. **Falso**, `Stream` è un'interfaccia.
2. **Falso**.
3. **Vero**.

4. **Vero**.
5. **Falso**, `Optional` è una classe (peraltro dichiarata `final` e quindi non estendibile).
6. **Falso**, sono operazioni terminali.
7. **Falso**, il metodo `joining()` appartiene alla classe `Collectors`.
8. **Falso**.
9. **Vero**.
10. **Falso**.

Soluzione 17.c)

L'eccezione personalizzata potrebbe essere la seguente:

```
package com.claudiodesio.eccezioni;

public class DuplicatoException extends RuntimeException {

    public DuplicatoException(Object elementoDuplicato) {
        super("Impossibile aggiungere l'elemento \""
          + elementoDuplicato + "\" perché già presente");
    }
}
```

Mentre potremmo codificare la collection richiesta in questo modo:

```
package com.claudiodesio.collections;

import com.claudiodesio.eccezioni.DuplicatoException;
import java.util.HashSet;

public class SetRobusto<E> extends HashSet<E> {

    @Override
    public boolean add(E e) {
        boolean result = super.add(e);
        if (!result) {
            throw new DuplicatoException(e);
        }
        return result;
    }
}
```

Ed ecco una classe di test:

```
package com.claudiodesio;

import com.claudiodesio.collections.SetRobusto;
import com.claudiodesio.eccezioni.DuplicatoException;

public class TestSetRobusto {

    public static void main(String args[]) {
        SetRobusto<String> set = getSetRobusto();
        try {
            set.add("Italia");
        } catch (DuplicatoException duplicatoException) {
            System.out.println(duplicatoException.getMessage());
        }
        System.out.println(set);
    }

    public static SetRobusto<String> getSetRobusto() {
        SetRobusto<String> set = new SetRobusto<>();
        set.add("Italia");
        set.add("Francia");
        set.add("Polonia");
        set.add("Germania");
        set.add("Inghilterra");
        set.add("Spagna");
        set.add("Grecia");
        set.add("Olanda");
        set.add("Portogallo");
        set.add("Belgio");
        return set;
    }
}
```

L'output sarà:

```
Impossibile aggiungere l'elemento "Italia" perché già presente
[Germania, Inghilterra, Francia, Belgio, Polonia, Olanda, Italia,
 Spagna, Grecia, Portogallo]
```

Soluzione 17.d)

Il listato della mappa richiesta potrebbe essere il seguente:

```
package com.claudiodesio.collections;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;

public class MappaIncrementale<K, V> extends HashMap<K, Collection<V>> {
```

```
    public void add(K key, V value) {
        if (this.get(key) == null) {
            Collection<V> collection = new ArrayList<>();
            collection.add(value);
            this.put(key, collection);
        } else {
            Collection<V> collection = this.get(key);
            collection.add(value);
        }
    }
}
```

Si noti che il metodo `add()` non è un override (il metodo per aggiungere coppie chiave-valore è il metodo `put()`) bensì un metodo ad hoc.
Segue una classe di test:

```
package com.claudiodesio.test;

import com.claudiodesio.collections.MappaIncrementale;
import com.claudiodesio.collections.SetRobusto;
import java.util.Iterator;

public class TestMappaIncrementale {

    public static void main(String args[]) {
        MappaIncrementale<Integer, String> mappa = new MappaIncrementale<>();
        riempiMappaIncrementale( mappa);
        System.out.println(mappa);
    }

    public static void riempiMappaIncrementale(
      MappaIncrementale<Integer, String> mappa) {
        SetRobusto<String> set = TestSetRobusto.getSetRobusto();
        int i = 1;
        int j = 1;
        Iterator<String> iterator = set.iterator();
        while (iterator.hasNext()) {
            if (i % 3 == 0) {
                j++;
            }
            String string = iterator.next();
            mappa.add(j, string);
            i++;
        }
    }
}
```

che genera il seguente output:

```
{1=[Germania, Inghilterra], 2=[Francia, Belgio, Polonia], 3=[Olanda,
 Italia, Spagna], 4=[Grecia, Portogallo]}
```

Soluzione 17.e)

Il listato della mappa richiesta potrebbe essere il seguente:

```
package com.claudiodesio.collections;

import java.util.Collection;

public class MappaIncrementaleRobusta<K, V> extends MappaIncrementale<K, V> {

    @Override
    public void add(K key, V value) {
        if (this.get(key) == null) {
            Collection<V> setRobusto = new SetRobusto<>();
            setRobusto.add(value);
            this.put(key, setRobusto);
        } else {
            Collection<V> setRobusto = this.get(key);
            setRobusto.add(value);
        }
    }
}
```

Si noti che questa volta il metodo `add()` è un override, e semplicemente cambiando l'`ArrayList` definito nella classe `MappaIncrementale` con un `SetRobusto` abbiamo risolto la situazione. Volendo potremmo fare un po' di refactoring su queste due classi in modo tale da migliorare il nostro codice. Per prima cosa ritorniamo alla classe `MappaIncrementale` e riscriviamo il metodo `add()` con dei piccoli accorgimenti:

```
package com.claudiodesio.collections;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;

public class MappaIncrementale<K, V> extends HashMap<K, Collection<V>> {

    public void add(K key, V value) {
        if (this.get(key) == null) {
            Collection<V> collection = getCollection();
            collection.add(value);
            this.put(key, collection);
        } else {
            Collection<V> arrayList = this.get(key);
```

```
            arrayList.add(value);
        }
    }

    protected Collection <V> getCollection() {
        return new ArrayList<>();
    }
}
```

In questo modo possiamo semplificare la sottoclasse:

```
package com.claudiodesio.collections;

import java.util.Collection;

public class MappaIncrementaleRobusta<K, V> extends MappaIncrementale<K, V> {

    @Override
    protected Collection<V> getCollection() {
        return new SetRobusto<>();
    }
}
```

Ed ottenere lo stesso risultato, senza duplicazioni di codice.
Segue la classe di test:

```
package com.claudiodesio.test;

import com.claudiodesio.collections.MappaIncrementale;
import com.claudiodesio.collections.MappaIncrementaleRobusta;
import com.claudiodesio.eccezioni.DuplicatoException;

public class TestMappaIncrementaleRobusta {

    public static void main(String args[]) {

        MappaIncrementale<Integer, String> mappa =
                        new MappaIncrementaleRobusta<>();
        TestMappaIncrementale.riempiMappaIncrementale(mappa);
        try {
            mappa.add(4, "Grecia");
        } catch (DuplicatoException duplicatoException) {
            System.out.println(duplicatoException.getMessage());
        }
        System.out.println(mappa);

    }
}
```

Che genera il seguente output:

```
Impossibile aggiungere l'elemento "Grecia" perché già presente
{1=[Germania, Inghilterra], 2=[Francia, Belgio, Polonia], 3=[Olanda,
  Italia, Spagna], 4=[Grecia, Portogallo]}
```

Soluzione 17.f)

Nella classe `Citta` aggiungiamo i metodi richiesti:

```
    @Override
    public int hashCode() {
        int hash = 7;
        hash = 19 * hash + Objects.hashCode(this.nome);
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Citta other = (Citta) obj;
        if (!Objects.equals(this.nome, other.nome)) {
            return false;
        }
        return true;
    }
```

La classe `TestStreams` invece potrebbe essere codificata come segue:

```
package com.claudiodesio.test;

import com.claudiodesio.collections.SetRobusto;
import com.claudiodesio.dati.Citta;

public class TestStreams {

    public static void main(String args[]) {
        SetRobusto<Citta> set = getSetRobusto();
        System.out.println("Città di mare:");
        set.stream().filter(e->e.isDiMare()).forEach(System.out::println);
        System.out.println("\nCittà capoluogo:");
        set.stream().filter(e->e.isCapoluogo()).forEach(System.out::println);
        System.out.println("\nCittà che finiscono con 'a':");
        set.stream().filter(e->e.getNome().endsWith("a")).
          forEach(System.out::println);
    }
```

```
    public static SetRobusto<Citta> getSetRobusto() {
        SetRobusto<Citta> set = new SetRobusto<>();
        set.add(new Citta("Milano", true, false));
        set.add(new Citta("Rovigo", false, false));
        set.add(new Citta("Potenza", true, false));
        set.add(new Citta("Siracusa", false, true));
        set.add(new Citta("Perugia", true, false));
        set.add(new Citta("Napoli", true, true));
        set.add(new Citta("Pescara", false, true));
        set.add(new Citta("Taranto", false, true));
        set.add(new Citta("Siena", false, false));
        return set;
    }
}
```

Segue l'output:

```
Città di mare:
Napoli
Siracusa
Taranto
Pescara

Città capoluogo:
Napoli
Potenza
Perugia
Milano

Città che finiscono con 'a':
Potenza
Perugia
Siena
Siracusa
Pescara
```

Soluzione 17.g) Ciclo for migliorato, Vero o Falso:

Le affermazioni corrette sono le numero 2, 4 e 5.

Soluzione 17.h)

La risposta corretta è la numero 3 in quanto `HashMap` non estende `Collection`, e quindi non si può assegnare un reference di tipo `Collection` ad un `HashMap`. L'output della compilazione è infatti il seguente:

```
Esercizio17H.java:5: warning: [rawtypes] found raw type: Collection
        Collection map = new HashMap(10);
```

```
        ^
  missing type arguments for generic class Collection<E>
  where E is a type-variable:
    E extends Object declared in interface Collection
Esercizio17H.java:5: warning: [rawtypes] found raw type: HashMap
        Collection map = new HashMap(10);
                             ^
  missing type arguments for generic class HashMap<K,V>
  where K,V are type-variables:
    K extends Object declared in class HashMap
    V extends Object declared in class HashMap
Esercizio17H.java:5: error: incompatible types: HashMap cannot be
  converted to Collection
        Collection map = new HashMap(10);
                         ^
Esercizio17H.java:6: error: cannot find symbol
        map.put(1,1);
            ^
  symbol:   method put(int,int)
  location: variable map of type Collection
2 errors
2 warnings
```

Si noti che vengono mostrati anche due warning perché abbiamo usato raw type, e che anche gli errori sono due, perché il reference `map`, essendo di tipo `Collection`, non dichiara il metodo `put()` (che viene dichiarato nell'interfaccia `Map`).

Soluzione 17.i)

Le risposte corrette sono le numero 1 e 3. `Iterator` non estende `Iterable`, quindi l'affermazione 2 è non corretta. Nel caso dell'affermazione 4 l'affermazione sarebbe stata corretta se fosse stata: `Collection` **estende** l'interfaccia `Iterable`. Infatti sia `Collection` che `Iterable` sono interfacce e non classi. Questo implica che una può estendere l'altra, non implementarla.

Soluzione 17.l)

L'unica risposta corretta è la numero 4. L'affermazione numero 1 è falsa semplicemente perché `Collection` è un'interfaccia e non una classe. Per quanto riguarda l'affermazione 2, il metodo `toArray()` è dichiarato dall'interfaccia `Collection`. Inoltre, non esiste un metodo `toCollection()` nella classe `Arrays`, quindi anche l'affermazione 3 non è corretta.

Soluzione 17.m)

L'output della classe Esercizio17M è il seguente:

```
*
$
$
*
```

Infatti vengono inseriti due elementi (le stringhe `*` e `@`) con il metodo `add()` nella lista `list`, e poi con il metodo `set()` viene sovrascritto la stringa `@` con la stringa `$`. I successivi cicli stampano gli elementi della lista ciclando con un `ListIterator` prima in avanti e poi all'indietro.

Soluzione 17.n)

La soluzione potrebbe essere la seguente classe:

```
import java.util.*;

public class Esercizio17N {
    public static void main(String args[]) {
        ArrayList<Integer> list = new ArrayList<>();
        list.ensureCapacity(50);
        long startTime = System.currentTimeMillis();
        for (int i = 1; i <=100000; ++i) {
            if (i%2==0) {
                list.add(i);
            }
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Tempo = " + (endTime - startTime));
    }
}
```

Soluzione 17.o)

Tutte le affermazioni sono corrette tranne la numero 1. Infatti essendo `Set` estesa da `SortedSet` (vedi affermazione numero 3), le implementazioni di `SortedSet` come `TreeSet` saranno ordinate ("sorted" in inglese significa "ordinato"). La numero 2 è vera proprio in virtù del fatto che qualsiasi `Set` non ammette duplicati, quindi non è possibile neanche aggiungere due volte l'elemento `null`.

Soluzione 17.p)

Solo le affermazioni numero 3 e 4 sono corrette.

Soluzione 17.q)

La soluzione potrebbe essere la seguente classe:

```
import java.util.*;

public class Esercizio17Q {

   public static void main(String args[]) {
      ClaudioLinkedList claudioLinkedList = new ClaudioLinkedList();
      claudioLinkedList.removeFirst();
      claudioLinkedList.addFirst("C");
      claudioLinkedList.set(2, "A");
      claudioLinkedList.removeLast();
      claudioLinkedList.addLast("O");
      System.out.println(claudioLinkedList);
   }
}
```

Soluzione 17.r)

Potremmo utilizzare un `Hashtable`, una `ConcurrentHashMap`, oppure utilizzare un reference che punta al risultato dell'invocazione di un metodo sincronizzatore `synchronizedMap()`.

Soluzione 17.s)

Potremmo utilizzare un reference che punta al risultato dell'invocazione di un metodo sincronizzatore `unmodifiableList()`, come riportato di seguito:

```
List<String> immutableList = Arrays.asList("a", "b", "c");
immutableList = Collections.unmodifiableList(immutableList);
```

Oppure è possibile utilizzare il metodo di convenienza statico dell'interfaccia `List of()`, introdotto in Java 9:

```
List immutableList = List.of("a", "b", "c");
```

Stesso discorso per le implementazioni immutabili di `Set`. Ecco i due esempi richiesti:

```
Set<String> immutableSet = new HashSet<>(Arrays.asList("a", "b", "c"));
immutableSet = Collections.unmodifiableSet(immutableSet);
```

e sfruttando il metodo statico `of()` dell'interfaccia `Set`:

```
Set<String> immutableSet = Set.of("a", "b", "c");
```

Soluzione 17.t)

Una possibile soluzione potrebbe essere la seguente:

```
import java.util.*;

public class Esercizio17T {

    public static void main(String args[]) {
        List<String> stringList = getStringList();
        Double average = stringList.stream().
        filter(s -> !s.startsWith("a")).mapToInt(
        String::length).average().getAsDouble();
        System.out.println(average);
    }

   public static List<String> getStringList() {
        String stringa = "I ragazzi che si amano si baciano in piedi "
          + "Contro le porte della notte "
          + "E i passanti che passano li segnano a dito "
          + "Ma i ragazzi che si amano "
          + "Non ci sono per nessuno "
          + "Ed è la loro ombra soltanto "
          + "Che trema nella notte "
          + "Stimolando la rabbia dei passanti "
          + "La loro rabbia il loro disprezzo le risa la loro invidia "
          + "I ragazzi che si amano non ci sono per nessuno "
          + "Essi sono altrove molto più lontano della notte "
          + "Molto più in alto del giorno "
          + "Nell'abbagliante splendore del loro primo amore ";
        String[] stringhe = stringa.split(" ");
        return Arrays.asList(stringhe);
    }
}
```

Che produrrà il seguente output:

```
4.27710843373494
```

Soluzione 17.u)

Le affermazioni corrette sono le numero 2 e 3. L'affermazione numero 1 è scorretta perché `Optional` è una classe generica. La 4 è scorretta perché `findFirst()` è dichiarata nell'interfaccia `Stream`.

Soluzione 17.v)

La soluzione potrebbe essere la seguente (la pipeline introdotta è in grassetto):

```
import java.util.*;
import java.util.stream.*;

public class Esercizio17V {
```

```
    public static void main(String args[]) {
        List<String> stringList = getStringList();
        Map<String, List<String>> map =
           stringList.stream().collect(Collectors.groupingBy(
           s -> (""+s.charAt(0)).toLowerCase()));
        System.out.println(map);
    }

    public static List<String> getStringList() {
        String stringa ="I ragazzi che si amano si baciano in piedi "
          + "Contro le porte della notte "
          + "E i passanti che passano li segnano a dito "
          + "Ma i ragazzi che si amano "
          + "Non ci sono per nessuno "
          + "Ed è la loro ombra soltanto "
          + "Che trema nella notte "
          + "Stimolando la rabbia dei passanti "
          + "La loro rabbia il loro disprezzo le risa la loro invidia "
          + "I ragazzi che si amano non ci sono per nessuno "
          + "Essi sono altrove molto più lontano della notte "
          + "Molto più in alto del giorno "
          + "Nell'abbagliante splendore del loro primo amore ";
        String[] stringhe = stringa.split(" ");
        return Arrays.asList(stringhe);
    }
}
```

Si noti che la `Function` passata al metodo `groupingBy()`, ritorna il primo carattere, che viene "sommato" ad una stringa vuota per essere tramutato in stringa, per poi essere reso minuscolo per evitare di avere distinzioni tra maiuscole e minuscole. Purtroppo l'output non è molto leggibile (vedi esercizio successivo):

```
{a=[amano, a, amano, amano, altrove, alto, amore], b=[baciano], c=[che,
 Contro, che, che, ci, Che, che, ci], d=[della, dito, dei, disprezzo,
 della, del, del], e=[E, Ed, Essi], g=[giorno], è=[è], i=[I, in, i, i,
 il, invidia, I, in], l=[le, li, la, loro, la, La, loro, loro, le, la,
 loro, lontano, loro], m=[Ma, molto, Molto], n=[notte, Non, nessuno,
 nella, notte, non, nessuno, notte, Nell'abbagliante], o=[ombra],
 p=[piedi, porte, passanti, passano, per, passanti, per, più, più,
 primo], r=[ragazzi, ragazzi, rabbia, rabbia, risa, ragazzi], s=[si, si,
 segnano, si, sono, soltanto, Stimolando, si, sono, sono, splendore],
 t=[trema]}
```

Soluzione 17.z)

La soluzione potrebbe essere la seguente (in grassetto la pipeline richiesta):

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

```
import java.util.stream.*;

public class Esercizio17Z {

    public static void main(String args[]) {
        List<String> stringList = getStringList();
        Map<String, List<String>> map = stringList.stream().collect(
            Collectors.groupingBy(s -> (""+s.charAt(0)).toLowerCase()));
        map.entrySet().stream().forEach(System.out::println);
    }

    public static List<String> getStringList() {
        String stringa ="I ragazzi che si amano si baciano in piedi "
          + "Contro le porte della notte "
          + "E i passanti che passano li segnano a dito "
          + "Ma i ragazzi che si amano "
          + "Non ci sono per nessuno "
          + "Ed è la loro ombra soltanto "
          + "Che trema nella notte "
          + "Stimolando la rabbia dei passanti "
          + "La loro rabbia il loro disprezzo le risa la loro invidia "
          + "I ragazzi che si amano non ci sono per nessuno "
          + "Essi sono altrove molto più lontano della notte "
          + "Molto più in alto del giorno "
          + "Nell'abbagliante splendore del loro primo amore ";
        String[] stringhe = stringa.split(" ");
        return Arrays.asList(stringhe);
    }
}
```

L'output è decisamente più chiaro:

```
a=[amano, a, amano, amano, altrove, alto, amore]
b=[baciano]
c=[che, Contro, che, che, ci, Che, che, ci]
d=[della, dito, dei, disprezzo, della, del, del]
e=[E, Ed, Essi]
g=[giorno]
è=[è]
i=[I, in, i, i, il, invidia, I, in]
l=[le, li, la, loro, la, La, loro, loro, le, la, loro, lontano, loro]
m=[Ma, molto, Molto]
n=[notte, Non, nessuno, nella, notte, non, nessuno, notte,
  Nell'abbagliante]
o=[ombra]
p=[piedi, porte, passanti, passano, per, passanti, per, più, più,
  primo]
r=[ragazzi, ragazzi, rabbia, rabbia, risa, ragazzi]
s=[si, si, segnano, si, sono, soltanto, Stimolando, si, sono, sono,
  splendore]
t=[trema]
```

Esercizi del capitolo 18

Input-Output

Gli esercizi di questo capitolo saranno inizialmente orientati alla teoria, con tre esercizi di tipo vero-falso. Poi realizzeremo in vari step una semplice rubrica per fare un po' di pratica sulla programmazione, non sono sull'Input-Output. La parte finale di questo documento è invece dedicato agli esercizi propedeutici alla certificazione da programmatore Oracle.

Esercizio 18.a) Input - Output, Vero o Falso:

1. Il pattern Decorator permette di implementare una sorta di ereditarietà dinamica. Questo significa che, invece di creare tante classi quante sono le entità da astrarre, al runtime sarà possibile concretizzare uno di questi concetti direttamente con un oggetto.
2. I reader e i writer permettono di leggere e scrivere caratteri. Per tale ragione sono detti Character Stream.
3. All'interno del package `java.io` l'interfaccia `Reader` ha il ruolo di `ConcreteComponent`.
4. All'interno del package `java.io` l'interfaccia `InputStream` ha il ruolo di `ConcreteDecorator`.
5. Un `BufferedWriter` è un `ConcreteDecorator`.
6. Gli stream che possono realizzare una comunicazione direttamente con una fonte o una destinazione vengono detti "node stream".

7. I node stream di tipo `OutputStream` possono utilizzare il metodo:

```
int write(byte cbuf[])
```

per scrivere su una destinazione.

8. Il seguente oggetto `in`:

```
BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
```

permette di usufruire di un metodo `readLine()` che leggerà frasi scritte con la tastiera delimitate dalla battitura del tasto Invio.

9. Il seguente codice:

```
File outputFile = new File("pippo.txt");
```

crea un file di testo chiamato pippo.txt nella cartella corrente.

10. Non è possibile decorare un `FileReader`.

Esercizio 18.b) Serializzazione, Vero o Falso:

1. Lo stato di un oggetto è definito dal valore delle sue variabili d'istanza (in un certo istante).

2. L'interfaccia `Serializable` non ha metodi.

3. `transient` è un modificatore applicabile a variabili e classi. Una variabile `transient` non viene serializzata con le altre variabili; una classe `transient` non è serializzabile.

4. `transient` è un modificatore applicabile a metodi e variabili. Una variabile `transient` non viene serializzata con le altre variabili; un metodo `transient` non è serializzabile.

5. Se si provasse a serializzare un oggetto che possiede tra le sue variabili d'istanza una variabile di tipo `Reader` dichiarata `transient`, otterremmo un `NotSerializableException` durante l'esecuzione.

6. Una variabile `static` non sarà serializzata.

7. Una variabile d'istanza di tipo `OuputStream` deve essere dichiarata `transient` affinché sia coinvolta in una serializzazione.

8. Non è possibile usare come parametro di un "try with resources" un'implementazione di `Serializable`.
9. È possibile creare un clone di un oggetto `transient` con metodi di input output.
10. Un oggetto può essere serializzato anche in `Base64`.

Esercizio 18.c) New Input Output, Vero o Falso:

1. NIO 2.0 sostituisce del tutto il modello di input output definito con il package `java.io`.
2. L'utilizzo della classe `File` potrebbe essere completamente rimpiazzato dall'utilizzo della classe `Files` e dell'interfaccia `Path`.
3. Il metodo `toPath()` della classe `File` restituisce l'oggetto `Path` equivalente.
4. `Path` è un'interfaccia perché la sua implementazione dipende dalla piattaforma.
5. Il metodo `relativize()` appartiene alla classe `Files` e restituisce il percorso per arrivare dal `Path` specificato come primo argomento al `Path` specificato come secondo argomento.
6. Il metodo `subPath()` dell'interfaccia `Path` non restituisce il nodo radice.
7. Il metodo `delete()` dell'interfaccia `Path` solleverà un'eccezione nel caso si tenti di eliminare una directory non vuota.
8. Non ha senso ottenere `Reader` o `Writer` da un oggetto `Files`.
9. Il nome del file temporaneo è sempre stabilito dal sistema operativo.
10. Un file temporaneo viene salvato in una directory che dipende dal sistema operativo.

Esercizio 18.d)

Creare una classe `EditorInterattivo` contenente un metodo `main()` che permetta di scrivere in un file quello che si scrive da riga di comando.

Esercizio 18.e)

Creare una classe `More` contenente un metodo `main()` che simuli il programma more di linux. Una volta eseguito aspetterà che l'utente specifichi il file da leggere

in input. Una volta caricato il file il programma rimarrà in attesa di un input utente. Se l'utente preme la lettera "n" seguita da "invio", allora verranno visualizzate le prime 10 righe del file caricato, ed ogni volta che viene ripetuta l'azione stamperà le successive 10 righe, fino alla fine del file. Specificando la lettera "q" seguita da "invio" il programma dovrà terminare.

Esercizio 18.f)

Riprendiamo l'esercizio G.b dell'appendice G dove viene richiesto di creare una rubrica. Di seguito viene riproposta la traccia per comodità.

Realizzare un'applicazione che simuli il funzionamento di una rubrica. Il lettore si limiti a simulare la seguente situazione: una rubrica contiene informazioni (nome, indirizzo, numero telefonico) su un certo numero di contatti prestabilito (per esempio 5). Queste informazioni possono essere pre-caricate quando l'applicazione parte. L'utente dovrà fornire all'applicazione un nome da riga di comando e l'applicazione dovrà restituire le informazioni relative al contatto. Se il nome non è fornito, o se il nome immesso non corrisponde al nome di una persona pre-introdotta dall'applicazione, deve essere restituito un messaggio significativo.

In questo e nei prossimi esercizi, realizzeremo la rubrica passo dopo passo, usando, come strato di persistenza dei dati, un componente che si occupa di serializzare e deserializzare oggetti sul file system. Quindi partiamo proprio da questo strato che si occupa della serializzazione e della deserializzazione dei dati. Fare in modo che siano esposti i metodi `inserisci()` e `recupera()` che poi la rubrica dovrà invocare. In particolare al metodo `inserisci()` bisognerà passare ciò che deve essere registrato dalla rubrica (un oggetto della classe `Contatto`), mentre a `recupera()` dovrà essere passato in input il nome del contatto da recuperare. Sarò necessario progettare anche la classe `Contatto` che rappresenta i dati che la rubrica dovrà registrare, visto che costituirà l'input e l'output dei metodi `inserisci()` e `recupera()`. Si crei anche una classe di test con la quale si possa verificare che i metodi creati funzionino correttamente. Questa classe deve semplicemente creare un array di tre contatti.

Esercizio 18.g)

Partendo dalla soluzione dell'esercizio precedente, aggiungere due metodi di test che testano:

1. cosa succede se si prova ad aggiungere un contatto già esistente;
2. cosa succede se si prova a recuperare un contatto inesistente.

Riflettete sui risultati dei test, e limitatevi solo a progettare le soluzioni degli eventuali problemi che sono stati riconosciuti.

Esercizio 18.h)

Partendo dalla soluzione dell'esercizio precedente, implementare le soluzioni progettate.

Esercizio 18.i)

Partendo dalla soluzione dell'esercizio precedente, aggiungiamo nella classe `GestioneFile` i metodi:

1. `modifica()` che modifica un contatto (non si dovrà poter cambiare il nome del contatto);
2. `rimuovi()` che cancella il contatto con il nome specificato.

Creare anche i metodi nella classe di test che testano il corretto funzionamento dei metodi `modifica()` e `rimuovi()`.

Esercizio 18.l)

Partendo dalla soluzione dell'esercizio precedente aggiungiamo un altro requisito, questa volta non funzionale ma architetturale: applicando il principio di inversione della dipendenza (DIP, introdotto nella nota del paragrafo 17.2.2.2 a pagina 593) creiamo un semplice strato astratto (classi astratte e/o interfacce) che rappresentino le specifiche delle classi che usiamo. Facciamo quindi evolvere la classe `GestoreFile`, e la classe `Contatto`, in modo tale che rappresentino un'implementazione di uno strumento generico per interagire con un meccanismo di immagazzinamento qualsiasi.
Creiamo quindi un'interfaccia `GestoreSerializzazione` da far implementare a `GestoreFile`, e un'interfaccia `Dato` da far implementare a `Contatto`. Questa modifica ci permetterà di cambiare facilmente il modo in cui serializziamo i nostri oggetti (vedi esercizio 18.n).

Esercizio 18.m)

Partendo dalla soluzione dell'esercizio precedente aggiungiamo un altro requisito. Facciamo un po' di refactoring del codice, eliminando il codice duplicato nelle classi create. In particolare creare la classe `Esercizio18M`, eliminando le duplicazioni di codice che erano presenti nella classe `Esercizion18L`.
Naturalmente il lettore è libero di aggiungere qualsiasi miglioramento al proprio codice in qualsiasi classe.

Nello sviluppo è molto importante ad un certo punto fermarsi a riflettere e a fare un po' di ordine nel proprio codice, senza cambiarne il funzionamento ma facendo refactoring. Questo a qualcuno può sembrare una perdita di tempo ma favorisce la qualità del codice, che più sarà alta minore sarà la probabilità di creare malfunzionamenti nel software. Il mio consiglio personale è quello di dedicare una percentuale giornaliera del tempo di sviluppo che varia dal 10% al 25% al refactoring, possibilmente nel momento in cui ci si rende conto di essere troppo stanchi. Per esempio in una giornata lavorativa da 8 ore, le ultime due (o almeno l'ultima) potrebbe essere dedicata a migliorare la qualità del software. Per informazioni sul refactoring potete:

- andare sul sito http://refactoring.com**;**

- comprare il libro "Refactoring" di Martin Fowler, come consigliato nella bibliografia dell'appendice Z (è un libro datato 1999, ma ancora molto utile);

- utilizzare i meccanismi di refactoring che offrono gli IDE più famosi.

Esercizio 18.n)

Partendo dalla soluzione dell'esercizio precedente, creiamo anche un'implementazione alternativa della classe `GestoreFile`, che fa uso della libreria NIO2. Implementare anche la classe di test `Esercizio18N`.

Esercizio 18.o)

Quali delle seguenti affermazioni sono corrette?

1. La classe `File` definisce un metodo `getPath()` che restituisce una stringa.
2. La classe `File` definisce un metodo `getParent()` che restituisce un oggetto `File` che rappresenta una directory.
3. `pathSeparator` è una costante statica della classe `File`, che rappresenta il separatore di path dipendente dal sistema operativo.
4. La classe `File` definisce un metodo `delete()` che restituisce un booleano `true` se il file sarà davvero eliminato, oppure `false` in caso contrario.

Esercizio 18.p)

Quali delle seguenti affermazioni sono corrette?

1. Le implementazioni di `InputStream` sono ottimizzate per leggere byte, le implementazioni di `OutputStream` sono ottimizzate per scrivere byte. Infatti vengono dette "byte stream".
2. Le classi alla base della gerarchia dei "character stream" sono `Reader` e `Writer`.
3. Il metodo `readLine()` è definito nella classe `Reader`.
4. Il costruttore di un `ObjectInputStream` deve prendere in input un oggetto di tipo `FileInputStream`.

Esercizio 18.q)

Quali delle seguenti affermazioni sono corrette?

1. Se istanziamo un oggetto `File` specificando come parametro del costruttore il nome di un file che non esiste, esso sarà creato.
2. Per leggere il contenuto di un file di testo, basta utilizzare la classe `FileReader`.
3. Una directory può essere creata grazie alla classe `File`.
4. La classe `File` definisce un metodo che restituisce i nomi dei file all'interno di una directory.

Esercizio 18.r)

Data la seguente classe:

```
import java.io.*;

public class Esercizio18R {
    public static void main(String args[]) throws IOException {
        try (FileOutputStream fos = new FileOutputStream("nuovo file.txt");
             DataOutputStream dos = new DataOutputStream(fos);) {
            dos.writeInt(8);
            dos.writeDouble(0.1176);
        }
    }
}
```

Una volta creato il file nuovo file.txt, quanto spazio occuperà sull'hard disk?

1. Non verrà creato nessun file perché il file non viene chiuso correttamente.
2. 12 byte.
3. Non verrà creato nessun file perché il file non compila.
4. 64 byte.
5. Non verrà creato nessun file perché durante l'esecuzione sarà lanciata un'eccezione.
6. 128 byte.
7. La dimensione del file dipende dalla piattaforma su cui viene eseguito il programma.
8. 96 byte.

Esercizio 18.s)

Data la seguente classe:

```
import java.io.*;

public class Esercizio18S {
    public static void main(String args[]) throws Exception {
        try (FileOutputStream fos = new FileOutputStream("nuovo file.txt");
             DataOutputStream dos = new DataOutputStream(fos);) {
            for (int i = 0; i < 100; i++) {
                dos.writeInt(i);
            }
        }
    }
}
```

Scrivere la classe che legge il file creato.

Esercizio 18.t)

Quali delle seguenti affermazioni sono corrette?

1. Per istanziare un `Reader` dobbiamo gestire una `IOException`.
2. Per scrivere da un `OutputStream` dobbiamo gestire una `IOException`.
3. Per leggere da un oggetto `InputStream` dobbiamo gestire una `IOException`.

4. Per istanziare un oggetto `Writer` dobbiamo gestire una `FileNotFoundException`.

Esercizio 18.u)

Quali delle seguenti affermazioni sono corrette?

1. È importante chiudere uno stream, altrimenti la memoria che occupa non sarà deallocata sino al termine del programma.
2. L'unica classe che mette a diposizione Java per leggere in maniera interattiva istruzioni da riga di comando, è la classe `Scanner`.
3. La classe `BufferedReader` dichiara un metodo chiamato `readLine()` che legge una riga di testo intera dalla fonte passata in input al `BufferedReader`.
4. La classe `BufferedReader` dichiara un metodo chiamato `lines()` che restituisce un oggetto `Stream<String>`.

Esercizio 18.v)

Quali delle seguenti affermazioni sono corrette?

1. `Path` definisce un metodo che restituisce i nomi dei file all'interno di una directory.
2. `Path` dichiara un metodo per recuperare la directory in cui è contenuto.
3. `Path` dichiara un metodo per recuperare l'autore (l'owner) del file.
4. `Path` è un'interfaccia.

Esercizio 18.z)

Quali delle seguenti affermazioni sono corrette?

1. La classe `Files` definisce un metodo che restituisce i nomi dei file all'interno di una directory.
2. Il metodo `find()` della classe `Files` restituisce uno `Stream` di `Path`.
3. `Files` e `Path` appartengono al package `java.nio2.file`.
4. `Files` è un'interfaccia.
5. È possibile copiare un file in un altro con il metodo `copy()`.

Soluzioni degli esercizi del capitolo 18

Soluzione 18.a) Input - Output, Vero o Falso:

1. **Vero**.
2. **Vero**.
3. **Falso**.
4. **Falso**.
5. **Vero**.
6. **Vero**.
7. **Vero**.
8. **Vero**.
9. **Falso**.
10. **Falso**.

Soluzione 18.b) Serializzazione, Vero o Falso:

1. **Vero**.
2. **Vero**.
3. **Falso**.

4. **Falso**.
5. **Falso**.
6. **Vero**.
7. **Vero**.
8. **Falso**.
9. **Vero**.
10. **Vero**.

Soluzione 18.c) New Input Output, Vero o Falso:

1. **Vero**.
2. **Vero**.
3. **Vero**.
4. **Vero**.
5. **Falso**.
6. **Vero**.
7. **Vero**.
8. **Falso**.
9. **Falso**.
10. **Vero**.

Soluzione 18.d)

Il listato dovrebbe essere simile al seguente:

```
package com.claudiodesio.io;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class EditorInterattivo {

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

```
        File file = new File("file.txt");
        try (Scanner scanner = new Scanner(System.in);
             FileWriter fileWriter = new FileWriter(file);) {
            String stringa = "";
            System.out.println("Digita qualcosa e batti enter, oppure scrivi "
              + "\"fine\" per terminare il programma");
            while (!(stringa = scanner.nextLine()).equals("fine")) {
                System.out.println("Hai digitato " + stringa);
                fileWriter.append(stringa);
                fileWriter.flush();
            }

        } catch (IOException ex) {
            ex.printStackTrace();
        }
        System.out.println("Fine programma!");
    }
}
```

Soluzione 18.e)

Il listato dovrebbe essere simile al seguente:

```
package com.claudiodesio.io;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class More {

    private final static int numeroRighe = 10;
    private static final String QUIT_COMMAND = "q";
    private static final String NEXT_COMMAND = "n";

    public static void main(String args[]) {
        String carattere;
        System.out.println("Digita il nome di un file e batti enter, oppure "
          + "scrivi \"" + QUIT_COMMAND + "\" per terminare il programma");
        try (Scanner in = new Scanner(System.in);) {
            String nomeFile;
            if (!(nomeFile = in.nextLine()).equals(QUIT_COMMAND)) {
                File file = new File(nomeFile);
                try (Scanner fileScanner = new Scanner(file)) {
                    System.out.println("File trovato! \"" + nomeFile + "\"");
                    while (!(carattere = in.nextLine()).equals(QUIT_COMMAND)) {
                        if (carattere.equals(NEXT_COMMAND)) {
                            for (int i = 0; fileScanner.hasNext() &&
                                 i < numeroRighe; i++) {
                                System.out.println(fileScanner.nextLine());
                            }
```

```
                        }
                    }
                } catch (FileNotFoundException exc) {
                    System.out.println("File non trovato!");
                }
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        System.out.println("Fine programma!");
    }
}
```

Segue un esempio di output:

```
java com.claudiodesio.io.More
Digita il nome di un file e batti enter, oppure scrivi "q" per
 terminare il programma
More.java
File trovato! "More.java"
n
package com.claudiodesio.io;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class More {

    private final static int numeroRighe = 10;
    private static final String QUIT_COMMAND = "q";
n
    private static final String NEXT_COMMAND = "n";

    public static void main(String args[]) {
        String carattere;
        System.out.println("Digita il nome di un file e batti enter,
  oppure " + "scrivi \"" + QUIT_COMMAND + "\" per terminare il
  programma");
        try (Scanner in = new Scanner(System.in);) {
            String nomeFile;
            if (!(nomeFile = in.nextLine()).equals(QUIT_COMMAND)) {
                File file = new File(nomeFile);
n
                try (Scanner fileScanner = new Scanner(file)) {
                    System.out.println("File trovato! \"" +
  nomeFile + "\"");
                    while (!(carattere =
  in.nextLine()).equals(QUIT_COMMAND)) {
                        if (carattere.equals(NEXT_COMMAND)) {
                            for (int i = 0; fileScanner.hasNext()
```

```
   && i < numeroRighe; i++) {
                                    System.out.println(
   fileScanner.nextLine());
                                }
                            }
                        }
                    } catch (FileNotFoundException exc) {
n
                        System.out.println("File non trovato!");
                    }
                }
            } catch (Exception ex) {
                ex.printStackTrace();
            }
            System.out.println("Fine programma!");
        }
}
q
Fine programma!
```

Soluzione 18.f)

Possiamo creare la classe `Contatto` che conterrà i dati che vogliamo registrare nella `Rubrica`:

```
import java.io.Serializable;

public class Contatto implements Dato {

    private static final long serialVersionUID = 8942402240056525661L;

    private String nome;

    private String indirizzo;

    private String numeroDiTelefono;

    public Contatto (String nome, String indirizzo, String numeroDiTelefono) {
        this.nome = nome;
        this.indirizzo = indirizzo;
        this.numeroDiTelefono = numeroDiTelefono;
    }

    public void setNumeroDiTelefono(String numeroDiTelefono) {
        this.numeroDiTelefono = numeroDiTelefono;
    }

    public String getNumeroDiTelefono() {
        return numeroDiTelefono;
```

```
    }

    public void setIndirizzo(String indirizzo) {
        this.indirizzo = indirizzo;
    }

    public String getIndirizzo() {
        return indirizzo;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }

    @Override
    public String toString(){
        return "Nome:\t" + nome + "\nIndirizzo:\t" + indirizzo
          + "\nTelefono:\t" + numeroDiTelefono;
    }
}
```

Se non ricordate a cosa serve il serialVersionUID, consultate il paragrafo 9.5.1.

Prima di scrivere questo codice, abbiamo superficialmente eseguito l'analisi dei casi d'uso e degli scenari. Abbiamo anche abbozzato un class diagram e qualche diagramma dinamico per valutare gli scenari. Si sono evidenziati alcuni problemi non specificati dal problem statement definito nella traccia dell'esercizio. Per esempio come si sarebbero dovuti chiamare i file creati, dove posizionarli e così via. Abbiamo quindi dovuto compiere delle scelte, e abbiamo deciso di chiamare i file con il nome del contatto con un suffisso .con. Abbiamo definito appositamente la classe `FileUtils` come segue:

```
public class FileUtils {

    public static final String SUFFIX = ".con";

    public static String getNomeFile(String nome) {
        return nome + SUFFIX;
    }
}
```

Poi abbiamo implementato la classe `GestoreFile`, che si occupa di implementare i due metodi richiesti, `inserisci()` e `recupera()`, nel seguente modo:

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

public class GestoreFile {

    public void inserisci(Contatto contatto) throws IOException {
        try (FileOutputStream fos =
             new FileOutputStream (
             new File(FileUtils.getNomeFile(contatto.getNome())));
             ObjectOutputStream s = new ObjectOutputStream (fos);) {
            s.writeObject (contatto);
            System.out.println("Contatto registrato!");
        }
    }

    public Contatto recupera(String nome)
      throws IOException, ClassNotFoundException {
        Contatto contatto = null;
        try (FileInputStream fis = new FileInputStream (
            new File(FileUtils.getNomeFile(nome)));
        ObjectInputStream ois = new ObjectInputStream (fis);) {
            contatto = (Contatto)ois.readObject();
            System.out.println("Contatto recuperato:\n" + contatto);
        }
        return contatto;
    }
}
```

Si noti che abbiamo utilizzato il costrutto try with resources in entrambi i metodi. Infine la classe di test richiesta:

```
public class Esercizio18F {

    private GestoreFile gestoreFile;

    private Contatto[] contatti;

    Esercizio18F() {
        contatti = getContatti();
        gestoreFile = new GestoreFile();
    }

    private void eseguiTest() {
        System.out.println("TESTIAMO LA CREAZIONE DEI TRE CONTATTI");
        creaContatti();
        System.out.println("RECUPERIAMO I TRE CONTATTI");
        recuperaContatti();
```

```
    }

    public void creaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Creazione contatto:\n" + contatto);
            creaContatto(contatto);
        }
    }

    public void recuperaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Recupero contatto: " + contatto.getNome());
            recuperaContatto(contatto.getNome());
        }
    }

    public void recuperaContatto(String nomeContatto) {
        try {
            gestoreFile.recupera(nomeContatto);
        }
        catch (Exception exc) {
            exc.printStackTrace();
        }
    }

    private void creaContatto(Contatto contatto) {
        try {
            gestoreFile.inserisci(contatto);
        }
        catch (Exception exc) {
            exc.printStackTrace();
        }
    }

    private Contatto[] getContatti() {
        Contatto contatto1 =
          new Contatto("Daniele","Via delle chitarre 1","01234560");
        Contatto contatto2 =
          new Contatto("Giovanni","Via delle scienze 2","0565432190");
        Contatto contatto3 =
          new Contatto("Ligeia","Via dei segreti 3","07899921");
        Contatto[] contatti = {contatto1, contatto2, contatto3};
        return contatti;
    }

    public static void main(String args[]) {
        Esercizio18F esercizio18F = new Esercizio18F();
        esercizio18F.eseguiTest();
    }
}
```

Che produrrà il seguente output:

```
Creazione contatto:
Nome: Daniele
Indirizzo:  Via delle chitarre 1
Telefono:   01234560
Contatto registrato!
Creazione contatto:
Nome: Giovanni
Indirizzo:  Via delle scienze 2
Telefono:   0565432190
Contatto registrato!
Creazione contatto:
Nome: Ligeia
Indirizzo:  Via dei segreti 3
Telefono:   07899921
Contatto registrato!
Recupero contatto: Daniele
Contatto recuperato:
Nome: Daniele
Indirizzo:  Via delle chitarre 1
Telefono:   01234560
Recupero contatto: Giovanni
Contatto recuperato:
Nome: Giovanni
Indirizzo:  Via delle scienze 2
Telefono:   0565432190
Recupero contatto: Ligeia
Contatto recuperato:
Nome: Ligeia
Indirizzo:  Via dei segreti 3
Telefono:   07899921
```

Soluzione 18.g)

Potremmo codificare i nuovi metodi come nel seguente file (in grassetto il codice aggiunto):

```
public class Esercizio18G {

    private GestoreFile gestoreFile;

    private Contatto[] contatti;

    Esercizio18G() {
        contatti = getContatti();
        gestoreFile = new GestoreFile();
    }
```

```
    private void eseguiTest() {
        System.out.println("TESTIAMO LA CREAZIONE DEI TRE CONTATTI");
        creaContatti();
        System.out.println("RECUPERIAMO I TRE CONTATTI");
        recuperaContatti();
        System.out.println(
          "TESTIAMO LA CREAZIONE DI UN CONTATTO GIÀ ESISTENTE");
        creaContattoEsistente();
        System.out.println("PROVIAMO A RECUPERARE UN CONTATTO NON ESISTENTE");
        recuperaContattoNonEsistente();
    }

    public void creaContattoEsistente() {
        try {
            gestoreFile.inserisci(contatti[0]);
        } catch (Exception exc) {
            exc.printStackTrace();
        }
    }

    public void recuperaContattoNonEsistente() {
        try {
            gestoreFile.recupera("Pippo");
        } catch (Exception exc) {
            exc.printStackTrace();
        }
    }

    public void creaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Creazione contatto:\n" + contatto);
            creaContatto(contatto);
        }
    }

    public void recuperaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Recupero contatto: " + contatto.getNome());
            recuperaContatto(contatto.getNome());
        }
    }

    public void recuperaContatto(String nomeContatto) {
        try {
            gestoreFile.recupera(nomeContatto);
        } catch (Exception exc) {
            exc.printStackTrace();
        }
    }

    private void creaContatto(Contatto contatto) {
```

```
            try {
                gestoreFile.inserisci(contatto);
            } catch (Exception exc) {
                exc.printStackTrace();
            }
        }

        private Contatto[] getContatti() {
            Contatto contatto1 =
              new Contatto("Daniele","Via delle chitarre 1","01234560");
            Contatto contatto2 =
              new Contatto("Giovanni","Via delle scienze 2","0565432190");
            Contatto contatto3 =
              new Contatto("Ligeia","Via dei segreti 3","07899921");
            Contatto[] contatti = {contatto1, contatto2, contatto3};
            return contatti;
        }

        public static void main(String args[]) {
            Esercizio18G esercizio18G = new Esercizio18G();
            esercizio18G.eseguiTest();
        }
    }
```

L'output risultante sarà:

```
TESTIAMO LA CREAZIONE DEI TRE CONTATTI
Creazione contatto:
Nome: Daniele
Indirizzo:  Via delle chitarre 1
Telefono:   01234560
Contatto registrato!
Creazione contatto:
Nome: Giovanni
Indirizzo:  Via delle scienze 2
Telefono:   0565432190
Contatto registrato!
Creazione contatto:
Nome: Ligeia
Indirizzo:  Via dei segreti 3
Telefono:   07899921
Contatto registrato!
RECUPERIAMO I TRE CONTATTI
Recupero contatto: Daniele
Contatto recuperato:
Nome: Daniele
Indirizzo:  Via delle chitarre 1
Telefono:   01234560
Recupero contatto: Giovanni
Contatto recuperato:
Nome: Giovanni
```

```
Indirizzo:  Via delle scienze 2
Telefono:   0565432190
Recupero contatto: Ligeia
Contatto recuperato:
Nome: Ligeia
Indirizzo:  Via dei segreti 3
Telefono:   07899921
TESTIAMO LA CREAZIONE DI UN CONTATTO GIÀ ESISTENTE
Contatto registrato!
PROVIAMO A RECUPERARE UN CONTATTO NON ESISTENTE
java.io.FileNotFoundException: Pippo.con (Impossibile trovare il file
  specificato)
      at java.base/java.io.FileInputStream.open0(Native Method)
      at java.base/java.io.FileInputStream.open(
  FileInputStream.java:196)     at
  java.base/java.io.FileInputStream.<init>(FileInputStream.java:139)
      at GestoreFile.recupera(GestoreFile.java:17)
      at Esercizio18G.recuperaContattoNonEsistente(Esercizio18G.java:34)
      at Esercizio18G.eseguiTest(Esercizio18G.java:20)
      at Esercizio18G.main(Esercizio18G.java:84)
```

Si noti che il test ha dimostrato che:

1. aggiungere un contatto esistente provoca la sovrascrittura del vecchio contatto;
2. recuperare un contatto inesistente provoca un'eccezione di tipo `FileNotFoundException`.
3. Entrambi i risultati dei test non ci soddisfano. Nel primo caso, non ci sembra corretto che il vecchio contatto sia sostituito senza neanche che l'utente ne venga a conoscenza. Potremmo pensare ad una semplice soluzione che preveda prima di recuperare l'eventuale contatto per controllare se esiste già, e poi di lanciare un'eccezione personalizzata che potrebbe chiamarsi `ContattoEsistenteException`. Eventualmente in futuro potremmo creare un metodo `modifica()` per modificare un contatto esistente.
4. Nel secondo caso sarebbe preferibile creare un'eccezione personalizzata che potremmo chiamare `ContattoInesistenteException`, contenente un messaggio leggibile, per evitare lo stack trace della `FileNotFoundException`.

Soluzione 18.h)

Per prima cosa definiamo le eccezioni in maniera semplicistica, segue l'eccezione `ContattoInesistenteException`:

```
import java.io.IOException;

public class ContattoInesistenteException extends IOException {

    private static final long serialVersionUID = 8942402240056525663L;

    public ContattoInesistenteException(String message) {
        super(message);
    }
}
```

E l'altra eccezione `ContattoEsistenteException`:

```
import java.io.IOException;

public class ContattoEsistenteException extends IOException {

    private static final long serialVersionUID = 8942402240056525662L;

    public ContattoEsistenteException(String message) {
        super(message);
    }
}
```

Abbiamo fatto estendere ad entrambe le classi l'eccezione `IOException`. Modifichiamo la classe `GestoreFile` nel seguente modo:

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

public class GestoreFile {

    public void inserisci(Contatto contatto)
      throws ContattoEsistenteException, FileNotFoundException, IOException {
        Contatto contattoEsistente = getContatto(contatto.getNome());
        if (contattoEsistente != null) {
            throw new ContattoEsistenteException(contatto.getNome()
              + ": contatto già esistente!");
        }
        try (FileOutputStream fos = new FileOutputStream (
             new File(FileUtils.getNomeFile(contatto.getNome())));
             ObjectOutputStream s = new ObjectOutputStream (fos);) {
            s.writeObject (contatto);
            System.out.println("Contatto registrato!");
        }
    }

    public Contatto recupera(String nome) throws ContattoInesistenteException {
        Contatto contatto = getContatto(nome);
        if (contatto == null) {
            throw new ContattoInesistenteException(
```

```
                nome +": contatto non trovato!");
        }
        return contatto;
    }

    private Contatto getContatto(String nome) {
        try (FileInputStream fis = new FileInputStream (
             new File(FileUtils.getNomeFile(nome)));
             ObjectInputStream ois = new ObjectInputStream (fis);) {
            Contatto contatto = (Contatto)ois.readObject();
            System.out.println("Contatto recuperato:\n"+ contatto);
            return contatto;
        } catch (Exception exc) {
            return null;
        }
    }
}
```

Si noti che abbiamo aggiunto il metodo privato `getContatto()`, che ritorna il contatto se esiste oppure `null` se non viene trovato. Tale metodo viene utilizzato sia dal metodo `inserisci()` che dal metodo `recupera()`, al fine di gestire le eventuali eccezioni. Infine modifichiamo anche la classe di test sostituendo, con la stampa del metodo `getMessage()` delle eccezioni, lo statement che stampava lo stack trace dell'eccezione (in grassetto le modifiche):

```
public class Esercizio18H {

    private GestoreFile gestoreFile;

    private Contatto[] contatti;

    Esercizio18H() {
        contatti = getContatti();
        gestoreFile = new GestoreFile();
    }

    private void eseguiTest() {
        System.out.println("TESTIAMO LA CREAZIONE DEI TRE CONTATTI");
        creaContatti();
        System.out.println("RECUPERIAMO I TRE CONTATTI");
        recuperaContatti();
        System.out.println(
          "TESTIAMO LA CREAZIONE DI UN CONTATTO GIÀ ESISTENTE");
        creaContattoEsistente();
        System.out.println("PROVIAMO A RECUPERARE UN CONTATTO NON ESISTENTE");
        recuperaContattoNonEsistente();
    }

    public void creaContattoEsistente() {
```

```
        try {
            gestoreFile.inserisci(contatti[0]);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void recuperaContattoNonEsistente() {
        try {
            gestoreFile.recupera("Pippo");
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void creaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Creazione contatto:\n" + contatto);
            creaContatto(contatto);
        }
    }

    public void recuperaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Recupero contatto: " + contatto.getNome());
            recuperaContatto(contatto.getNome());
        }
    }

    public void recuperaContatto(String nomeContatto) {
        try {
            gestoreFile.recupera(nomeContatto);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    private void creaContatto(Contatto contatto) {
        try {
            gestoreFile.inserisci(contatto);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    private Contatto[] getContatti() {
        Contatto contatto1 =
          new Contatto("Daniele","Via delle chitarre 1","01234560");
        Contatto contatto2 =
          new Contatto("Giovanni","Via delle scienze 2","0565432190");
        Contatto contatto3 =
```

```
            new Contatto("Ligeia","Via dei segreti 3","07899921");
        Contatto[] contatti = {contatto1, contatto2, contatto3};
        return contatti;
    }

    public static void main(String args[]) {
        Esercizio18H esercizio18H = new Esercizio18H();
        esercizio18H.eseguiTest();
    }
}
```

L'output di questa classe adesso risulterà essere:

```
Creazione contatto:
Nome: Daniele
Indirizzo:  Via delle chitarre 1
Telefono:   01234560
Contatto recuperato:
Nome: Daniele
Indirizzo:  Via delle chitarre 1
Telefono:   01234560
Daniele: contatto già esistente!
Creazione contatto:
Nome: Giovanni
Indirizzo:  Via delle scienze 2
Telefono:   0565432190
Contatto recuperato:
Nome: Giovanni
Indirizzo:  Via delle scienze 2
Telefono:   0565432190
Giovanni: contatto già esistente!
Creazione contatto:
Nome: Ligeia
Indirizzo:  Via dei segreti 3
Telefono:   07899921
Contatto recuperato:
Nome: Ligeia
Indirizzo:  Via dei segreti 3
Telefono:   07899921
Ligeia: contatto già esistente!
Recupero contatto: Daniele
Contatto recuperato:
Nome: Daniele
Indirizzo:  Via delle chitarre 1
Telefono:   01234560
Recupero contatto: Giovanni
Contatto recuperato:
Nome: Giovanni
Indirizzo:  Via delle scienze 2
Telefono:   0565432190
Recupero contatto: Ligeia
```

```
Contatto recuperato:
Nome: Ligeia
Indirizzo:  Via dei segreti 3
Telefono:   07899921
Contatto recuperato:
Nome: Daniele
Indirizzo:  Via delle chitarre 1
Telefono:   01234560
Daniele: contatto già esistente!
Pippo: contatto non trovato!
```

Soluzione 18.i)

Coerentemente con la linea di sviluppo tenuta sinora, il file `GestoreFile`, potrebbe evolversi nel seguente modo (in grassetto le modifiche apportate):

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

public class GestoreFile {

    public void inserisci(Contatto contatto)
      throws ContattoEsistenteException, FileNotFoundException, IOException {
        Contatto contattoEsistente = getContatto(contatto.getNome());
        if (contattoEsistente != null) {
            throw new ContattoEsistenteException(
              contatto.getNome() +": contatto già esistente!");
        }
        registra(contatto);
    }

    public Contatto recupera(String nome)
      throws ContattoInesistenteException, ContattoEsistenteException {

        Contatto contatto = getContatto(nome);
        if (contatto == null) {
            throw new ContattoInesistenteException(
              nome + ": contatto non trovato!");
        }
        return contatto;
    }

    public void rimuovi(String nome) throws ContattoInesistenteException,
      ContattoEsistenteException, FileNotFoundException, IOException {
        File file = new File(FileUtils.getNomeFile(nome));
        if (file.delete()) {
            System.out.println("Contatto " + nome + " cancellato!");
        } else {
            throw new ContattoInesistenteException(
```

```
                nome + ": contatto non trovato!");
        }
    }

    private void registra(Contatto contatto)
      throws  FileNotFoundException, IOException {
        try (FileOutputStream fos =
             new FileOutputStream (
             new File(FileUtils.getNomeFile(contatto.getNome())));
             ObjectOutputStream s = new ObjectOutputStream (fos);) {
            s.writeObject (contatto);
            System.out.println("Contatto registrato:\n"+ contatto);
        }
    }

    private boolean isContattoEsistente(String nome) {
        File file = new File(FileUtils.getNomeFile(nome));
        return file.exists();
    }

    private Contatto getContatto(String nome) {
        try (FileInputStream fis = new FileInputStream (
             new File(FileUtils.getNomeFile(nome)));
             ObjectInputStream ois = new ObjectInputStream (fis);) {
            Contatto contatto = (Contatto)ois.readObject();
            System.out.println("Contatto recuperato:\n"+ contatto);
            return contatto;
        } catch (Exception exc) {
            return null;
        }
    }
}
```

Si noti che il metodo privato `isContattoEsistente()` controlla solo se un file con il nome del contatto esiste, e questo nel nostro caso dovrebbe essere sufficiente. Chiamiamo questo metodo dal metodo `modifica()` prima di registrare (e sovrascrivere) il vecchio contatto. Nel caso il file non esistesse, otterremmo il lancio di un'eccezione `ContattoInesistenteException`. Il metodo `rimuovi()` invece sfrutta il metodo `delete()` della classe `File`. La classe di test potremmo modificarla nel seguente modo, per andare a testare gli scenari della modifica e della cancellazione (in grassetto le modifiche apportate):

```
public class Esercizio18I {

    private GestoreFile gestoreFile;

    private Contatto[] contatti;

    Esercizio18I() {
```

```
        contatti = getContatti();
        gestoreFile = new GestoreFile();
    }

    private void eseguiTest() {
        System.out.println("TESTIAMO LA CREAZIONE DEI TRE CONTATTI");
        creaContatti();
        System.out.println("RECUPERIAMO I TRE CONTATTI");
        recuperaContatti();
        System.out.println(
          "TESTIAMO LA CREAZIONE DI UN CONTATTO GIÀ ESISTENTE");
        creaContattoEsistente();
        System.out.println("PROVIAMO A RECUPERARE UN CONTATTO NON ESISTENTE");
        recuperaContattoNonEsistente();
        System.out.println("MODIFICHIAMO UN CONTATTO ESISTENTE");
        modificaContattoEsistente();
        System.out.println("RIMUOVIAMO UN CONTATTO ESISTENTE");
        rimuoviContattoEsistente();
        System.out.println("MODIFICHIAMO UN CONTATTO NON ESISTENTE");
        modificaContattoNonEsistente();
        System.out.println("RIMUOVIAMO UN CONTATTO NON ESISTENTE");
        rimuoviContattoNonEsistente();
    }

    public void creaContattoEsistente() {
        try {
            gestoreFile.inserisci(contatti[0]);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void modificaContattoEsistente() {
        try {
            gestoreFile.modifica(
              new Contatto("Daniele","Via dei microfoni 1","07890"));
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void rimuoviContattoEsistente() {
        try {
            gestoreFile.rimuovi(contatti[2].getNome());
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void modificaContattoNonEsistente() {
        try {
```

```
            gestoreFile.modifica(new Contatto(
                "Pluto","Via dei microfoni 1","07890"));
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void rimuoviContattoNonEsistente() {
        try {
            gestoreFile.rimuovi("Ligeia");
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void recuperaContattoNonEsistente() {
        try {
            gestoreFile.recupera("Pippo");
        }
        catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void creaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Creazione contatto:\n"+ contatto);
            creaContatto(contatto);
        }
    }

    public void recuperaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Recupero contatto: "+ contatto.getNome());
            recuperaContatto(contatto.getNome());
        }
    }

    public void recuperaContatto(String nomeContatto) {
        try {
            gestoreFile.recupera(nomeContatto);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    private void creaContatto(Contatto contatto) {
        try {
            gestoreFile.inserisci(contatto);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
```

```
        }
    }

    private Contatto[] getContatti() {
        Contatto contatto1 =
          new Contatto("Daniele","Via delle chitarre 1","01234560");
        Contatto contatto2 =
          new Contatto("Giovanni","Via delle scienze 2","0565432190");
        Contatto contatto3 =
          new Contatto("Ligeia","Via dei segreti 3","07899921");
        Contatto[] contatti = {contatto1, contatto2, contatto3};
        return contatti;
    }

    public static void main(String args[]) {
        Esercizio18I Esercizio18I = new Esercizio18I();
        Esercizio18I.eseguiTest();
    }
}
```

L'output risultante sarà il seguente:

```
TESTIAMO LA CREAZIONE DEI TRE CONTATTI
Creazione contatto:
Nome: Daniele
Indirizzo:  Via delle chitarre 1
Telefono:   01234560
Contatto recuperato:
Nome: Daniele
Indirizzo:  Via dei microfoni 1
Telefono:   07890
Daniele: contatto già esistente!
Creazione contatto:
Nome: Giovanni
Indirizzo:  Via delle scienze 2
Telefono:   0565432190
Contatto recuperato:
Nome: Giovanni
Indirizzo:  Via delle scienze 2
Telefono:   0565432190
Giovanni: contatto già esistente!
Creazione contatto:
Nome: Ligeia
Indirizzo:  Via dei segreti 3
Telefono:   07899921
Contatto registrato:
Nome: Ligeia
Indirizzo:  Via dei segreti 3
Telefono:   07899921
RECUPERIAMO I TRE CONTATTI
Recupero contatto: Daniele
```

```
Contatto recuperato:
Nome: Daniele
Indirizzo:  Via dei microfoni 1
Telefono:   07890
Recupero contatto: Giovanni
Contatto recuperato:
Nome: Giovanni
Indirizzo:  Via delle scienze 2
Telefono:   0565432190
Recupero contatto: Ligeia
Contatto recuperato:
Nome: Ligeia
Indirizzo:  Via dei segreti 3
Telefono:   07899921
TESTIAMO LA CREAZIONE DI UN CONTATTO GIÀ ESISTENTE
Contatto recuperato:
Nome: Daniele
Indirizzo:  Via dei microfoni 1
Telefono:   07890
Daniele: contatto già esistente!
PROVIAMO A RECUPERARE UN CONTATTO NON ESISTENTE
Pippo: contatto non trovato!
MODIFICHIAMO UN CONTATTO ESISTENTE
Contatto registrato:
Nome: Daniele
Indirizzo:  Via dei microfoni 1
Telefono:   07890
RIMUOVIAMO UN CONTATTO ESISTENTE
Contatto Ligeia cancellato!
MODIFICHIAMO UN CONTATTO NON ESISTENTE
Pluto: contatto non trovato!
RIMUOVIAMO UN CONTATTO NON ESISTENTE
Ligeia: contatto non trovato!
```

Soluzione 18.l)

Per prima cosa creiamo una semplice interfaccia `Dato`:

```
import java.io.Serializable;

public interface Dato extends Serializable {
}
```

La classe `Contatto` la implementerà:

```
import java.io.Serializable;

public class Contatto implements Dato {

    private static final long serialVersionUID = 894240224005652566 1L;
```

```
    private String nome;

    private String indirizzo;

    private String numeroDiTelefono;

    public Contatto(String nome, String indirizzo, String numeroDiTelefono) {
        this.nome = nome;
        this.indirizzo = indirizzo;
        this.numeroDiTelefono = numeroDiTelefono;
    }

    public void setNumeroDiTelefono(String numeroDiTelefono) {
        this.numeroDiTelefono = numeroDiTelefono;
    }

    public String getNumeroDiTelefono() {
        return numeroDiTelefono;
    }

    public void setIndirizzo(String indirizzo) {
        this.indirizzo = indirizzo;
    }

    public String getIndirizzo() {
        return indirizzo;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }

    @Override
    public String toString() {
        return "Nome:\t" + nome + "\nIndirizzo:\t" + indirizzo +
          "\nTelefono:\t" + numeroDiTelefono;
    }
}
```

Quindi potremmo creare la seguente classe astratta `GestoreSerializzazione`:

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

public interface GestoreSerializzazione<T extends Dato> {
```

```
    void inserisci(T dato) throws IOException;

    T recupera(String id) throws IOException, ClassNotFoundException;

    void modifica(T dato) throws IOException;

    void rimuovi(String id) throws IOException;
}
```

per farla implementare a `GestoreFile`:

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

public class GestoreFile implements GestoreSerializzazione<Contatto> {

    @Override
    public void inserisci(Contatto contatto)
      throws ContattoEsistenteException, FileNotFoundException, IOException {
        Contatto contattoEsistente = getContatto(contatto.getNome());
        if (contattoEsistente != null) {
            throw new ContattoEsistenteException(
              contatto.getNome() +": contatto già esistente!");
        }
        registra( contatto);
    }

    @Override
    public Contatto recupera(String nome)
      throws ContattoInesistenteException, ContattoEsistenteException {
        Contatto contatto = getContatto(nome);
        if (contatto == null) {
            throw new ContattoInesistenteException(
              nome + ": contatto non trovato!");
        }
        return contatto;
    }

    @Override
    public void modifica(Contatto contatto)
      throws ContattoInesistenteException, ContattoEsistenteException,
      FileNotFoundException, IOException {
        if (isContattoEsistente(contatto.getNome())) {
            registra(contatto);
        } else {
            throw new ContattoInesistenteException(
              contatto.getNome() +": contatto non trovato!");
        }
    }
```

```
    @Override
    public void rimuovi(String nome)
      throws ContattoInesistenteException, ContattoEsistenteException,
      FileNotFoundException, IOException {
        File file = new File(FileUtils.getNomeFile(nome));
        if (file.delete()) {
            System.out.println("Contatto " + nome + " cancellato!");
        } else {
            throw new ContattoInesistenteException(
              nome + ": contatto non trovato!");
        }
    }

    private void registra(Contatto contatto)
      throws FileNotFoundException, IOException {
        try (FileOutputStream fos = new FileOutputStream (
             new File(FileUtils.getNomeFile(contatto.getNome())));
             ObjectOutputStream s = new ObjectOutputStream (fos);) {
            s.writeObject (contatto);
            System.out.println("Contatto registrato:\n" + contatto);
        }
    }

    private boolean isContattoEsistente(String nome) {
        File file = new File(FileUtils.getNomeFile(nome));
        return file.exists();
    }

    private Contatto getContatto(String nome) {
        try (FileInputStream fis = new FileInputStream (
             new File(FileUtils.getNomeFile(nome)));
             ObjectInputStream ois = new ObjectInputStream (fis);) {
            Contatto contatto = (Contatto)ois.readObject();
            System.out.println("Contatto recuperato:\n"+ contatto);
            return contatto;
        } catch (Exception exc) {
            return null;
        }
    }
}
```

Infine ecco la classe di test:

```
public class Esercizio18L {

    private GestoreSerializzazione<Contatto> gestoreFile;

    private Contatto[] contatti;

    Esercizio18L() {
        contatti = getContatti();
```

```
        gestoreFile = new GestoreFile();
    }

    private void eseguiTest() {
        System.out.println("TESTIAMO LA CREAZIONE DEI TRE CONTATTI");
        creaContatti();
        System.out.println("RECUPERIAMO I TRE CONTATTI");
        recuperaContatti();
        System.out.println(
          "TESTIAMO LA CREAZIONE DI UN CONTATTO GIÀ ESISTENTE");
        creaContattoEsistente();
        System.out.println("PROVIAMO A RECUPERARE UN CONTATTO NON ESISTENTE");
        recuperaContattoNonEsistente();
        System.out.println("MODIFICHIAMO UN CONTATTO ESISTENTE");
        modificaContattoEsistente();
        System.out.println("RIMUOVIAMO UN CONTATTO ESISTENTE");
        rimuoviContattoEsistente();
        System.out.println("MODIFICHIAMO UN CONTATTO NON ESISTENTE");
        modificaContattoNonEsistente();
        System.out.println("RIMUOVIAMO UN CONTATTO NON ESISTENTE");
        rimuoviContattoNonEsistente();    }

    public void creaContattoEsistente() {
        try {
            gestoreFile.inserisci(contatti[0]);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void modificaContattoEsistente() {
        try {
            gestoreFile.modifica(
              new Contatto("Daniele","Via dei microfoni 1","07890"));
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void rimuoviContattoEsistente() {
        try {
            gestoreFile.rimuovi(contatti[2].getNome());
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void modificaContattoNonEsistente() {
        try {
            gestoreFile.modifica(new Contatto(
                "Pluto","Via dei microfoni 1","07890"));
```

```
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void rimuoviContattoNonEsistente() {
        try {
            gestoreFile.rimuovi("Ligeia");
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void recuperaContattoNonEsistente() {
        try {
            gestoreFile.recupera("Pippo");
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void creaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Creazione contatto:\n" + contatto);
            creaContatto(contatto);
        }
    }

    public void recuperaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Recupero contatto: " + contatto.getNome());
            recuperaContatto(contatto.getNome());
        }
    }

    public void recuperaContatto(String nomeContatto) {
        try {
            gestoreFile.recupera(nomeContatto);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    private void creaContatto(Contatto contatto) {
        try {
            gestoreFile.inserisci(contatto);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }
```

```
    private Contatto[] getContatti() {
        Contatto contatto1 =
          new Contatto("Daniele","Via delle chitarre 1","01234560");
        Contatto contatto2 =
          new Contatto("Giovanni","Via delle scienze 2","0565432190");
        Contatto contatto3 =
          new Contatto("Ligeia","Via dei segreti 3","07899921");
        Contatto[] contatti = {contatto1, contatto2, contatto3};
        return contatti;
    }

    public static void main(String args[]) {
        Esercizio18L esercizio18L = new Esercizio18L();
        esercizio18L.eseguiTest();
    }
}
```

Soluzione 18.m)

Una soluzione potrebbe essere quella di usare delle espressioni lambda per evitare di riscrivere i vari metodi gestendo sempre allo stesso modo le eccezioni. In particolare possiamo creare due tipologie di interfacce funzionali. Una che definisce un metodo che restituisce `void`:

```
@FunctionalInterface
public interface Executor {

    void esegui() throws Exception;
}
```

Ed un'altra che restituisce un tipo generico:

```
@FunctionalInterface
public interface Retriever<O> {

    O esegui() throws Exception;
}
```

Possiamo sfruttare queste interfacce funzionali nella classe `Esercizio18M` nel seguente modo (le parti modificate come al solito sono in grassetto):

```
import java.util.function.*;

public class Esercizio18M {

    private GestoreSerializzazione<Contatto> gestoreFile;

    private Contatto[] contatti;
```

```
    Esercizio18M() {
        contatti = getContatti();
        gestoreFile = new GestoreFile();
    }

    private void eseguiTest() {
        System.out.println("TESTIAMO LA CREAZIONE DEI TRE CONTATTI");
        creaContatti();
        System.out.println("RECUPERIAMO I TRE CONTATTI");
        recuperaContatti();
        System.out.println(
          "TESTIAMO LA CREAZIONE DI UN CONTATTO GIÀ ESISTENTE");
        creaContattoEsistente();
        System.out.println("PROVIAMO A RECUPERARE UN CONTATTO NON ESISTENTE");
        recuperaContattoNonEsistente();
        System.out.println("MODIFICHIAMO UN CONTATTO ESISTENTE");
        modificaContattoEsistente();
        System.out.println("RIMUOVIAMO UN CONTATTO ESISTENTE");
        rimuoviContattoEsistente();
        System.out.println("MODIFICHIAMO UN CONTATTO NON ESISTENTE");
        modificaContattoNonEsistente();
        System.out.println("RIMUOVIAMO UN CONTATTO NON ESISTENTE");
        rimuoviContattoNonEsistente();
    }

    public void creaContattoEsistente() {
        esegui(()->gestoreFile.inserisci(contatti[0]));
    }

    public void modificaContattoEsistente() {
        esegui(()->gestoreFile.modifica(
          new Contatto("Daniele","Via dei microfoni 1","07890")));
    }

    public void rimuoviContattoEsistente() {
        esegui(()->gestoreFile.rimuovi(contatti[2].getNome()));
    }

    public void modificaContattoNonEsistente() {
        esegui(()->gestoreFile.modifica(
          new Contatto("Pluto","Via dei microfoni 1","07890")));
    }

    public void rimuoviContattoNonEsistente() {
        esegui(()->gestoreFile.rimuovi("Ligeia"));
    }

    public void recuperaContattoNonEsistente() {
        esegui(()->gestoreFile.recupera("Pippo"));
    }
```

```
    public void creaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Creazione contatto:\n" + contatto);
            creaContatto(contatto);
        }
    }

    public void recuperaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Recupero contatto: " + contatto.getNome());
            recuperaContatto(contatto.getNome());
        }
    }

    public void recuperaContatto(String nomeContatto) {
        esegui(()->gestoreFile.recupera(nomeContatto));
    }

    private void creaContatto(Contatto contatto) {
        esegui(()->gestoreFile.inserisci(contatto));
    }

    private Contatto[] getContatti() {
        Contatto contatto1 =
          new Contatto("Daniele","Via delle chitarre 1","01234560");
        Contatto contatto2 =
          new Contatto("Giovanni","Via delle scienze 2","0565432190");
        Contatto contatto3 =
          new Contatto("Ligeia","Via dei segreti 3","07899921");
        Contatto[] contatti = {contatto1, contatto2, contatto3};
        return contatti;
    }

    public <O> O esegui(Retriever<O> retriever) {
        O output = null;
        try {
            output = retriever.esegui();
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
        return output;
    }

    public void esegui(Executor executor) {
        try {
            executor.esegui();
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }
```

```
    public static void main(String args[]) {
        Esercizio18M esercizio18M = new Esercizio18M();
        esercizio18M.eseguiTest();
    }
}
```

Si noti che abbiamo dovuto creare i due metodi `esegui()` di cui uno prende in input un `Retriever` e l'altro un `Executor`. Questi metodi inseriscono la chiamata al metodo `esegui()` di `Retriever` o `Executor` all'interno della gestione delle eccezioni che prima veniva replicata in ogni metodo. In questo modo, passando un'espressione lambda contenente il codice da eseguire a questi due nuovi metodi, abbiamo potuto evitare le duplicazioni presenti nella classe `Esercizio18L`.

Soluzione 18.n)

La soluzione potrebbe essere la seguente:

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

public class GestoreFileNIO2 implements GestoreSerializzazione<Contatto> {

    @Override
    public void inserisci(Contatto contatto)
      throws ContattoEsistenteException, FileNotFoundException, IOException {
        Path path = Paths.get(FileUtils.getNomeFile(contatto.getNome()));
        if (Files.exists(path)) {
            throw new ContattoEsistenteException(
              contatto.getNome() + ": contatto già esistente!");
        }
        registra(contatto);
    }

    @Override
    public Contatto recupera(String nome)
      throws ContattoInesistenteException, ContattoEsistenteException {
        Contatto contatto = getContatto(nome);
        if (contatto == null) {
            throw new ContattoInesistenteException(
              nome + ": contatto non trovato!");
        }
        return contatto;
    }

    @Override
    public void modifica(Contatto contatto)
      throws ContattoInesistenteException, ContattoEsistenteException,
      FileNotFoundException, IOException {
```

```
        if (isContattoEsistente(contatto.getNome())) {
            registra(contatto);
        } else {
            throw new ContattoInesistenteException(contatto.getNome()
            + ": contatto non trovato!");
        }
    }

    @Override
    public void rimuovi(String nome) throws ContattoInesistenteException,
      ContattoEsistenteException, FileNotFoundException, IOException {
        Path path = Paths.get(FileUtils.getNomeFile(nome));
        if (Files.exists(path)) {
            Files.delete(path);
            System.out.println("Contatto " + nome + " cancellato!");
        } else {
            throw new ContattoInesistenteException(nome
            + ": contatto non trovato!");
        }
    }

    private void registra(Contatto contatto)
      throws  FileNotFoundException, IOException {
        Path path = Paths.get(FileUtils.getNomeFile(contatto.getNome()));
        Files.write(path, getBytesDaOggetto(contatto));
        System.out.println("Contatto registrato:\n" + contatto);
    }

    private byte[] getBytesDaOggetto(Object object) throws IOException {
        try (ByteArrayOutputStream bos = new ByteArrayOutputStream();
             ObjectOutput out = new ObjectOutputStream(bos)) {
            out.writeObject(object);
            return bos.toByteArray();
        }
    }

    private Object getOggettoDaByte(byte[] bytes)
      throws IOException, ClassNotFoundException {
        try (ByteArrayInputStream bis = new ByteArrayInputStream(bytes);
             ObjectInput in = new ObjectInputStream(bis)) {
            return in.readObject();
        }
    }

    private boolean isContattoEsistente(String nome) {
        Path path = Paths.get(FileUtils.getNomeFile(nome));
        return Files.exists(path);
    }

    private Contatto getContatto(String nome) {
        Path path = Paths.get(FileUtils.getNomeFile(nome));
```

```
            byte[] bytes = null;
            Contatto contatto = null;
            try {
                bytes = Files.readAllBytes(path);
                contatto = (Contatto)getOggettoDaByte(bytes);
                System.out.println("Contatto recuperato:\n" + contatto);
            } catch (Exception exc) {
                return null;
            }
            return contatto;
        }
    }
```

La classe per testare `Esercizio18N`, si differenzia dalla classe `Esercizio18M` solo dall'applicazione del principio DIP che istanzia la nuova classe `GestorFileNIO2`, usando un reference di tipo `GestoreSerializzazione`. Tutto il resto non cambia.

```
import java.util.function.*;

public class Esercizio18N {

    private GestoreSerializzazione<Contatto> gestoreFile;

    private Contatto[] contatti;

    Esercizio18N() {
        contatti = getContatti();
        gestoreFile = new GestoreFileNIO2();
    }

    private void eseguiTest() {
        System.out.println("TESTIAMO LA CREAZIONE DEI TRE CONTATTI");
        creaContatti();
        System.out.println("RECUPERIAMO I TRE CONTATTI");
        recuperaContatti();
        System.out.println(
          "TESTIAMO LA CREAZIONE DI UN CONTATTO GIÀ ESISTENTE");
        creaContattoEsistente();
        System.out.println("PROVIAMO A RECUPERARE UN CONTATTO NON ESISTENTE");
        recuperaContattoNonEsistente();
        System.out.println("MODIFICHIAMO UN CONTATTO ESISTENTE");
        modificaContattoEsistente();
        System.out.println("RIMUOVIAMO UN CONTATTO ESISTENTE");
        rimuoviContattoEsistente();
        System.out.println("MODIFICHIAMO UN CONTATTO NON ESISTENTE");
        modificaContattoNonEsistente();
        System.out.println("RIMUOVIAMO UN CONTATTO NON ESISTENTE");
        rimuoviContattoNonEsistente();
    }
```

```
    public void creaContattoEsistente() {
        esegui(()->gestoreFile.inserisci(contatti[0]));
    }

    public void modificaContattoEsistente() {
        esegui(()->gestoreFile.modifica(new Contatto(
          "Daniele","Via dei microfoni 1","07890")));
    }

    public void rimuoviContattoEsistente() {
        esegui(()->gestoreFile.rimuovi(contatti[2].getNome()));
    }

    public void modificaContattoNonEsistente() {
        esegui(()->gestoreFile.modifica(new Contatto(
          "Pluto","Via dei microfoni 1","07890")));
    }

    public void rimuoviContattoNonEsistente() {
        esegui(()->gestoreFile.rimuovi("Ligeia"));
    }

    public void recuperaContattoNonEsistente() {
        esegui(()->gestoreFile.recupera("Pippo"));
    }

    public void creaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Creazione contatto:\n" + contatto);
            creaContatto(contatto);
        }
    }

    public void recuperaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Recupero contatto: " + contatto.getNome());
            recuperaContatto(contatto.getNome());
        }
    }

    public void recuperaContatto(String nomeContatto) {
        esegui(()->gestoreFile.recupera(nomeContatto));
    }

    private void creaContatto(Contatto contatto) {
        esegui(()->gestoreFile.inserisci(contatto));
    }

    private Contatto[] getContatti() {
        Contatto contatto1 =
```

```
            new Contatto("Daniele","Via delle chitarre 1","01234560");
        Contatto contatto2 =
            new Contatto("Giovanni","Via delle scienze 2","0565432190");
        Contatto contatto3 =
            new Contatto("Ligeia","Via dei segreti 3","07899921");
        Contatto[] contatti = {contatto1, contatto2, contatto3};
        return contatti;
    }

    public <O> O esegui(Retriever<O> retriever) {
        O output = null;
        try {
            output = retriever.esegui();
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
        return output;
    }

    public void esegui(Executor executor) {
        try {
            executor.esegui();
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public static void main(String args[]) {
        Esercizio18N esercizio18N = new Esercizio18N();
        esercizio18N.eseguiTest();
    }
}
```

Possiamo ritornare alla vecchia versione semplicemente sostituendo all'istanza di `GestioneFileNIO2` l'istanza di `GestioneFile`.

Soluzione 18.o)

Le affermazioni corrette sono la prima e la quarta. L'affermazione numero 2 è scorretta in quanto `getParent()` restituisce una stringa che rappresenta una directory. L'affermazione numero 3 è scorretta in quanto `pathSeparator` è una variabile statica della classe `File`, che rappresenta il separatore di path dipendente dal sistema operativo. Lo si poteva evincere sia dal nome che è scritto in minuscolo (le costanti invece per convenzione sono scritte in maiuscolo), sia dal fatto che non può assumere un valore a priori visto che esso dipende dal sistema operativo.

Soluzione 18.p)

L'unica affermazione corretta è la prima. L'affermazione numero 2 è scorretta in quanto `Reader` e `Writer` sono interfacce e non classi. L'affermazione numero 3 è scorretta in quanto il metodo `readLine()` è definito all'interno della classe `BufferedReader`. L'affermazione numero 4 è scorretta in quanto il costruttore di un `ObjectInputStream` può prendere in input un qualsiasi oggetto di tipo `InputStream`.

Soluzione 18.q)

L'unica affermazione scorretta è la numero 1. L'affermazione numero 2 è corretta anche se l'efficienza e la comodità della lettura verrebbero indubbiamente migliorate "agganciando" un `BufferedReader` al `FileReader`. Per esempio potremmo leggere un file con un codice simile al seguente:

```
Reader fileReader = new FileReader("c:/miofile.txt");
int data = fileReader.read();
while (data != -1) {
    System.out.print(data);
    data = fileReader.read();
}
fileReader.close();
```

L'affermazione numero 3 è corretta, perché la classe `File` definisce il metodo `mkDir()`, ed anche la 4 è corretta in quanto la classe `File` definisce il metodo `list()`.

Soluzione 18.r)

La risposta corretta è la numero 2, ovvero 12 byte. Infatti vengono scritti all'interno del file un tipo `int` (32 bit = 3 byte) e un `double` (64 bit = 8 byte).

Soluzione 18.s)

La soluzione potrebbe essere la seguente:

```
import java.io.*;

public class Soluzione18S {
    public static void main(String args[]) throws Exception {
        try (FileInputStream fis = new FileInputStream("nuovo file.txt");
             DataInputStream dis = new DataInputStream(fis);) {
            for (int i = 0; i < 50; i++) {
                System.out.print(dis.readInt());
            }
        }
```

```
        }
    }
```

Il cui output è:

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
```

```
46
47
48
49
```

Soluzione 18.t)

Le affermazioni corrette sono le numero 2 e 3. La 1 non è corretta perché esistono implementazioni di `Reader`, come `StringReader`, che non dichiarano clausole `throws` a `IOException`. L'affermazione 4 non è corretta in quanto devono gestire tali implementazioni solo le implementazioni di `Writer` che hanno a che fare con i file come `FileWriter`.

Soluzione 18.u)

Le affermazioni corrette sono le numero 1, 3 e 4. L'affermazione 2 non è corretta perché abbiamo visto, anche nel paragrafo 18.4.1, che è possibile ottenere lo stesso risultato concatenando un `BufferedReader` ad uno `InputStreamReader` la cui fonte di lettura è l'oggetto `System.in` (ovvero il dispositivo di input di default, probabilmente la tastiera). Per comodità riportiamo di seguito la classe definita nel paragrafo 18.4.1 `KeyboardInput`:

```
import java.io.*;

public class KeyboardInput {

    public static void main (String args[]) throws IOException {
        String stringa = null;
        System.out.println("Digita qualcosa e premi invio...\n"
          + "Per terminare il programma digitare \"fine\"");
        try (InputStreamReader ir = new InputStreamReader(System.in);
             BufferedReader in = new BufferedReader(ir)) {
            stringa = in.readLine();
            while ( stringa != null ) {
                if (stringa.equals("fine")) {
                    System.out.println("Programma terminato");
                    break;
                }
                System.out.println("Hai scritto: " + stringa);
                stringa = in.readLine();
            }
        }
    }
}
```

Soluzione 18.v)

Solo le affermazioni 1 e 3 non sono corrette. In particolare l'affermazione 2 è corretta alla luce dell'esistenza del metodo `getParent()`. La 3 invece non è corretta visto che il metodo `getOwner()` è definito dalla classe `Files`.

Soluzione 18.z)

Solo le affermazioni 2 e 5 sono corrette. L'affermazione 1 non è corretta perché, il metodo `list()` di `Files`, restituisce uno `Stream` di oggetti `Path`. La 3 non è corretta perché i tipi `Files` e `Path` appartengono al package `java.nio.file`. Mentre la 4 non è corretta perché `Files` è una classe astratta.

Esercizi del capitolo 19

Moduli

Non sarà facile iniziare ad utilizzare i moduli per i programmatori. Si tratta di un concetto che abbraccia una branca dell'informatica diversa: l'architettura software. Gli esercizi presentati di seguito puntano prima a chiarire tutti i concetti teorici, per poi creare moduli in maniera progressiva sino a creare un'architettura sostenibile, per l'applicazione che simula una rubrica creata con gli esercizi del modulo precedente.

Esercizio 19.a)

Quali delle eseguenti affermazioni sono corrette:

1. Con il sistema modulare possiamo evitare eccezioni al runtime come `ClassNotFoundException`.
2. Il forte incapsulamento permette di rendere accessibile un certo package solo ai package specificati.
3. Nel JDK 9 il file rt.jar è stato eliminato.
4. La JVM della versione 9 quando esegue un programma che usa i moduli deve gestirli, e quindi risulterà meno performante rispetto a quando eseguirà un programma che non utilizza i moduli.
5. Possiamo evitare di utilizzare i moduli perché esiste il concetto di modulo anonimo.

Esercizio 19.b)

Quali delle seguenti affermazioni sono vere?

1. Un file JAR modulare ha estensione .jmod.
2. Il JDK 9 ha definito nuove applicazioni come jlink e molte altre applicazioni come jdeps del JDK sono state modificate per supportare l'introduzione dei moduli.
3. Un modulo è costituito da un unico file.
4. Le classi di alcuni package come `sun.*`, e `*.internal.*` sono state eliminate.
5. Un modulo automatico è un file JAR puntato dal module path.

Esercizio 19.c)

Quali delle seguenti affermazioni sono vere?

1. Le parole che definiscono le direttive del modulo (`module`, `open`, `opens`, `provides`, `requires`, `to`, `transitive`, `uses` e `with`) sono dette "restricted word".
2. Sia i package sia i moduli seguono la stessa convenzione per i nomi.
3. È possibile annotare il descrittore di un modulo module-info.java con alcune annotazioni.
4. Il descrittore di un modulo module-info.class deve trovarsi nella directory radice del modulo.

Esercizio 19.d)

Spiegare cosa significa eseguire il seguente comando:

```
javac -d mods/com.domain.mymodule src/com.domain.mymodule/com/domain/*
  src/com.domain.mymodule/module-info.java
```

Esercizio 19.e)

Spiegare cosa significa eseguire il seguente comando:

```
java --module-path mods -m com.domain/com.domain.HelloModularWorld
```

Esercizio 19.f)

Definire i seguenti concetti riguardante l'architettura software:

1. Componente software
2. Partizionamento verticale
3. Framework (partizionamento orizzontale)
4. Coesione
5. Accoppiamento
6. Sottosistema
7. Principio di inversione della dipendenza

Esercizio 19.g)

Immaginiamo di avere creato un software gestionale per un negozio di computer. Immaginiamo anche di avere un modulo che contiene il codice che gestisce la fatturazione del negozio, e chiamiamolo `negozio.fatturazione`. Questo modulo contiene i package`negozio.fatturazione.documenti`,`negozio.fatturazione.algoritmiinterni` e `negozio.fatturazione.funzionidisponibili`. Poi consideriamo un altro modulo che contiene il codice che rappresenta l'interfaccia grafica dell'applicazione, e chiamiamolo `negozio.interfacciagrafica`. Dichiarare i relativi descrittori che esplicitano la dipendenza che esiste tra questi due moduli.

Esercizio 19.h)

Partendo dalla soluzione dell'esercizio precedente dichiarare il descrittore di un modulo che contiene il codice che permetta la vendita di articoli chiamato `negozio.vendita`, ed eventualmente modificare i descrittori già definiti. Tenere presente che:

1. Il modulo `negozio.vendita` contiene i package chiamati `negozio.vendita.funzionidisponibili`,`negozio.vendita.articoli` e `negozio.vendita.algoritmiinterni`.
2. Dall'interfaccia grafica sarà possibile vendere articoli del negozio.
3. Contestualmente alla vendita deve essere possibile fatturare l'articolo venduto.

Esercizio 19.i)

Partendo dalla soluzione dell'esercizio precedente, cosa possiamo fare se volessimo che il modulo `negozio.fatturazione` legga `negozio.interfacciagrafica`?

Esercizio 19.l)

Partendo dalla soluzione dell'esercizio 19.h, cosa possiamo fare per permettere al modulo `negozio.interfacciagrafica` di accedere tramite reflection ad un metodo privato definito nel package `negozio.fatturazione.algoritmiinterni`?

Esercizio 19.m)

Consideriamo la soluzione dell'esercizio precedente. Se non è possibile modificare più il nostro codice, ma ci siamo accorti che una classe del package `negozio.vendita.algoritmiinterni` deve utilizzare un metodo privato del package `negozio.fatturazione.algoritmiinterni` tramite reflection, cosa possiamo fare?

Esercizio 19.n)

Quali delle seguenti affermazioni è corretta:

1. La classe `ServiceLoader` è stata introdotta già nella versione 6 di Java.
2. Il componente service provider interface (SPI) dipende dalle sue implementazioni.
3. Con la classe `ServiceLoader` possiamo eliminare completamente la dipendenza tra moduli.
4. Per implementare un servizio con `ServiceLoader`, le implementazioni di una service provider interface devono essere esportate dai rispettivi moduli.
5. Un metodo provider è una sorta di metodo factory.

Esercizio 19.o)

Partendo dalla soluzione dell'esercizio 18.n, definire dei package per le varie classi. Poi creare e compilare un modulo che esporta il package che contiene tutte le classi che rappresentano dati.

Esercizio 19.p)

Partendo dalla soluzione dell'esercizio 19.o, creare un modulo che esporti il package che contiene le eccezioni.

Per compilare con un unico comando più moduli solitamente si utilizza l'attributo --module-source-path. Per esempio per compilare l'esercizio del paragrafo 19.3 sul `ServiceLoader`, abbiamo usato il seguente comando:

```
javac -d mods --module-source-path
  src src/com.claudiodesio.spi/module-info.java
  src/com.claudiodesio.spi/com/claudiodesio/spi/*
  src/com.claudiodesio.invs/module-info.java
  src/com.claudiodesio.invs/com/claudiodesio/invs/*
  src/com.claudiodesio.certs/module-info.java
  src/com.claudiodesio.certs/com/claudiodesio/certs/*
  src/com.claudiodesio.factory/module-info.java
  src/com.claudiodesio.factory/com/claudiodesio/factory/*
  src/com.claudiodesio.handlers/module-info.java
  src/com.claudiodesio.handlers/com/claudiodesio/handlers/*
  src/com.claudiodesio.client/module-info.java
  src/com.claudiodesio.client/com/claudiodesio/client/*
```

(scritto tutto su una riga senza andare a capo).

Esercizio 19.q)

Partendo dalla soluzione dell'esercizio 19.p, si crei un modulo che esporti il package contenente le classi di utilità.

Esercizio 19.r)

Partendo dalla soluzione dell'esercizio 19.q, creare un modulo che esporti il package contenente la classe che gioca il ruolo di service provider interface. Il nostro obiettivo è quello di trasformare il modo in cui serializziamo i contatti con servizi di `ServiceLoader`.

Esercizio 19.s)

Riportiamo di seguito la classe `Esercizio18N` che rappresenta il client della rubrica creata negli esercizi del capitolo 18:

```
import java.util.function.*;

public class Esercizio18N {

    private GestoreSerializzazione<Contatto> gestoreFile;

    private Contatto[] contatti;

    Esercizio18N() {
        contatti = getContatti();
```

```
            gestoreFile = new GestoreFileNIO2();
        }

        private void eseguiTest() {
            System.out.println("TESTIAMO LA CREAZIONE DEI TRE CONTATTI");
            creaContatti();
            System.out.println("RECUPERIAMO I TRE CONTATTI");
            recuperaContatti();
            System.out.println(
                "TESTIAMO LA CREAZIONE DI UN CONTATTO GIÀ ESISTENTE");
            creaContattoEsistente();
            System.out.println("PROVIAMO A RECUPERARE UN CONTATTO NON ESISTENTE");
            recuperaContattoNonEsistente();
            System.out.println("MODIFICHIAMO UN CONTATTO ESISTENTE");
            modificaContattoEsistente();
            System.out.println("RIMUOVIAMO UN CONTATTO ESISTENTE");
            rimuoviContattoEsistente();
            System.out.println("MODIFICHIAMO UN CONTATTO NON ESISTENTE");
            modificaContattoNonEsistente();
            System.out.println("RIMUOVIAMO UN CONTATTO NON ESISTENTE");
            rimuoviContattoNonEsistente();
        }

        public void creaContattoEsistente() {
            esegui(()->gestoreFile.inserisci(contatti[0]));
        }

        public void modificaContattoEsistente() {
            esegui(()->gestoreFile.modifica(new Contatto(
                "Daniele","Via dei microfoni 1","07890")));
        }

        public void rimuoviContattoEsistente() {
            esegui(()->gestoreFile.rimuovi(contatti[2].getNome()));
        }

        public void modificaContattoNonEsistente() {
            esegui(()->gestoreFile.modifica(
                new Contatto("Pluto","Via dei microfoni 1","07890")));
        }

        public void rimuoviContattoNonEsistente() {
            esegui(()->gestoreFile.rimuovi("Ligeia"));
        }

        public void recuperaContattoNonEsistente() {
            esegui(()->gestoreFile.recupera("Pippo"));
        }

        public void creaContatti() {
            for (Contatto contatto : contatti) {
```

```
            System.out.println("Creazione contatto:\n"+ contatto);
            creaContatto(contatto);
        }
    }

    public void recuperaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Recupero contatto: "+ contatto.getNome());
            recuperaContatto(contatto.getNome());
        }
    }

    public void recuperaContatto(String nomeContatto) {
        esegui(()->gestoreFile.recupera(nomeContatto));
    }

    private void creaContatto(Contatto contatto) {
        esegui(()->gestoreFile.inserisci(contatto));
    }

    private Contatto[] getContatti() {
        Contatto contatto1 = new Contatto(
            "Daniele","Via delle chitarre 1","01234560");
        Contatto contatto2 = new Contatto(
            "Giovanni","Via delle scienze 2","0565432190");
        Contatto contatto3 = new Contatto(
            "Ligeia","Via dei segreti 3","07899921");
        Contatto[] contatti = {
            contatto1, contatto2, contatto3
        } ;
        return contatti;
    }

    public <O> O esegui(Retriever<O> retriever) {
        O output = null;
        try {
            output = retriever.esegui();
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
        return output;
    }

    public void esegui(Executor executor) {
        try {
            executor.esegui();
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }
```

```
    public static void main(String args[]) {
        Esercizio18N esercizio18N = new Esercizio18N();
        esercizio18N.eseguiTest();
    }
}
```

Abbiamo riportato in grassetto le istruzioni su cui concentrare la nostra attenzione. Si noti come nel costruttore viene creato esplicitamente un oggetto della classe `GestoreNIO2`, ed assegnato ad un reference di tipo `GestoreSerializzazione`. Il nostro obiettivo per questo esercizio sarà quello di creare due moduli diversi per trasformare in servizi i due modi in cui abbiamo serializzato gli oggetti `Contatto` nel capitolo 18 `GestoreFile` e `GestoreNIO2`. Poi nel prossimo esercizio li utilizzeremo.

Esercizio 19.t)

Partendo dalla soluzione dell'esercizio 19.s, rinominare la classe `Esercizio18N` in `Esercizio19T`, e sfruttare il `ServiceLoader` per caricare i servizi. Fare in modo che sia possibile specificare da riga di comando quale gestore di serializzazione si deve usare. Poi creare un modulo che contenga tale classe ed eseguire l'applicazione.

Esercizio 19.u)

Partendo dalla soluzione dell'esercizio 19.t, creare una classe factory e spostare al suo interno il metodo `getGestoreSerializzazione()` presente nella classe `Esercizio19T` e il relativo modulo. Modificare di conseguenza anche il descrittore del modulo `com.claudiodesio.rubrica.test`.

Esercizio 19.v)

Impacchettare i moduli creati nei rispettivi JAR modulari ed eseguire l'applicazione.

Esercizio 19.z)

Con jlink creare un ambiente personalizzato con il solo modulo `java.base`. Poi utilizzando la cartella lib dell'esercizio precedente eseguire il nostro JAR modulare eseguibile tramite il runtime appena creato.

Soluzioni degli esercizi del capitolo 19

Soluzione 19.a)

Le affermazioni corrette sono le numero 1, 3 e 5. L'affermazione numero 2 è scorretta perché il forte incapsulamento permette di rendere accessibile un certo package solo ai moduli specificati (non ai package specificati). L'affermazione numero 4 è scorretta perché come affermato nel paragrafo 19.1.1, le tecniche di ottimizzazione della JVM sono più efficaci nel caso siano noti a priori i tipi che saranno utilizzati nell'applicazione.

Soluzione 19.b)

Le uniche affermazioni corrette sono le numero 2 e 5. La numero 1 non è corretta perché un JAR modulare ha un'estensione .jar. Un file con estensione .jmod invece è caratterizzato dal fatto di contenere anche risorse native. L'affermazione numero 3 è palesemente falsa: ad essere un unico file è il descrittore del modulo, non il modulo stesso. Infine l'affermazione 4 è anch'essa falsa perché i package in questione non sono stati eliminati, bensì nascosti all'utilizzo mediante il forte incapsulamento.

Soluzione 19.c)

Tutte le affermazioni sono corrette.

Soluzione 19.d)

Con il seguente comando:

```
javac -d mods/com.domain.mymodule src/com.domain.mymodule/com/domain/*
  src/com.domain.mymodule/module-info.java
```

stiamo compilando un modulo di nome `com.domain.mymodule`.
In particolare con l'opzione:

```
-d mods/com.domain.mymodule
```

stiamo ordinando al comando javac che il risultato della compilazione dovrà essere posizionato all'interno della sottocartella com.domain.mymodule della cartella mods (cartella che deve essere presente nella stessa posizione da dove stiamo eseguendo il comando di compilazione).
Con l'argomento:

```
src/com.domain.mymodule/com/domain/*
```

stiamo invece specificando che devono essere compilati tutti i file presenti nel percorso src/com.domain.mymodule/com/domain.
Infine con l'argomento:

```
src/com.domain.mymodule/module-info.java
```

specifichiamo che deve essere compilato anche il descrittore del modulo module-info.java presente nella cartella src/com.domain.mymodule.

Soluzione 19.e)

Con il seguente comando:

```
java --module-path mods -m com.domain/com.domain.mymodule.HelloModularWorld
```

stiamo eseguendo la classe contenente il metodo `main()` `com.domain.mymodule.HelloModularWorld` del modulo di nome `com.domain.mymodule`, specificando come module path la cartella mods.

Soluzione 19.f)

Un **componente software** è un insieme di classi con un'interfaccia ben definita, che mette a disposizione funzionalità eseguibili indipendentemente dal contesto. Un componente software è quindi un sottosistema eseguibile e riutilizzabile.
In un **partizionamento verticale** l'applicazione viene divisa per funzionalità della stessa importanza, e ognuna di queste può essere sviluppata indipendentemente dall'altra.

Un **partizionamento orizzontale** è un partizionamento basato soprattutto sull'ereditarietà. Un tipico esempio è quello di un **framework**, ovvero una micro-architettura che mette a disposizione un modello estendibile per realizzare applicazioni nell'ambito di uno specifico dominio.
La **coesione** è la misura di quanto un certo elemento (classe, metodo, package, sottosistema, etc.) contribuisce a realizzare un certo scopo all'interno del sistema.
L'**accoppiamento** è la metrica che misura la dipendenza tra classi, tra package, tra metodi e così via.
Un **sottosistema** è un insieme di classi relazionate da associazioni, eventi e vincoli, e per il quale è possibile uno sviluppo indipendente dagli altri sottosistemi. I sottosistemi sono altamente coesi (ovvero dichiara solo tipi che collaborano tra loro per realizzare un determinato scopo), internamente sono presenti tipi altamente accoppiati, esternamente sono bassamente accoppiati con tipi di altre classi.
Il **principio di inversione della dipendenza** afferma che le classi devono dipendere dalle astrazioni e non dalle implementazioni.

Soluzione 19.g)

Una soluzione potrebbe essere la seguente:

```
module negozio.interfacciagrafica {
    requires negozio.fatturazione;
}
```

e

```
module negozio.fatturazione {
    exports negozio.fatturazione.documenti;
    exports negozio.fatturazione.funzionidisponibili;
    //exports negozio.fatturazione.algoritmiinterni;
}
```

Ovvero il modulo `negozio.interfacciagrafica` legge il modulo `negozio.fatturazione`, che a sua volta gli espone solo due dei tre package (si noti che la terza direttiva è commentata). Infatti il package `negozio.fatturazione.algoritmiinterni`, considerato il nome, con tutta probabilità potrebbe non essere utilizzato direttamente dall'esterno.

Soluzione 19.h)

Muovendoci in un contesto tanto astratto, è possibile ipotizzare diverse soluzioni. Partendo dal presupposto che il descrittore del modulo `negozio.fatturazione` non sarà modificato, potremmo semplicemente esportare i giusti package dal mo-

dulo `negozio.vendita`:

```
module negozio.vendita {
    exports negozio.vendita.articoli;
    exports negozio.vendita.funzionidisponibili;
    //exports negozio.vendita.algoritmiinterni;
}
```

e far in modo che il modulo `negozio.interfacciagrafica`, legga anche `negozio.vendita`:

```
module negozio.interfacciagrafica {
    requires negozio.fatturazione;
    requires negozio.vendita;
}
```

delegando così all'interfaccia grafica l'onere di definire una funzione che vada a contestualizzare la vendita e la fatturazione di un articolo in un'unica richiesta da parte dell'utente. Siccome non è una buona pratica assegnare delle regole di business ad un'interfaccia grafica (che deve già definire le regole di presentazione dell'interfaccia stessa), decidiamo di rendere la situazione più flessibile, in modo tale da poter prendere delle decisioni più tardi. Allora utilizzando la direttiva `requires transitive`, andiamo a modificare il descrittore del modulo `negozio.vendita`, in modo tale che possa leggere (e far leggere) il modulo `negozio.fatturazione`:

```
module negozio.vendita {
    exports negozio.vendita.articoli;
    exports negozio.vendita.funzionidisponibili;
    //exports negozio.vendita.algoritmiinterni;
    requires transitive negozio.fatturazione;
}
```

A questo punto il modulo `negozio.interfacciagrafica` può leggere solo `negozio.vendita`, visto che transitivamente leggerà anche `negozio.fatturazione`.

```
module negozio.interfacciagrafica {
    //requires negozio.fatturazione;
    requires negozio.vendita;
}
```

Soluzione 19.i)

In realtà non si può fare nulla a meno che non si vogliano rivalutare tutte le dipendenze già specificate. Se `negozio.fatturazione` volesse leggere `negozio.interfacciagrafica`, otterremmo un errore di dipendenza ciclica.

Per quelli che sono gli standard architetturali più utilizzati (vedi MVC) in effetti non c'è nessuna ragione per cui l'interfaccia grafica di un programma debba essere letta da un modulo di business.

Soluzione 19.l)

La soluzione consiste nel rivedere il descrittore del modulo `negozio.fatturazione` esportando il package `negozio.vendita.algoritmiinterni`, e contemporaneamente aprirlo al modulo `negozio.interfacciagrafica`:

```
module negozio.fatturazione {
    exports negozio.fatturazione.articoli;
    exports negozio.fatturazione.funzionidisponibili;
    exports negozio.fatturazione.algoritmiinterni;
    opens negozio.fatturazione.algoritmiinterni to negozio.interfacciagrafica;
}
```

Soluzione 19.m)

L'unica soluzione è eseguire il programma specificando da riga di comando l'apertura del modulo `negozio.fatturazione` al modulo `negozio.vendita` con la seguente sintassi:

```
-add-opens negozio.fatturazione/negozio.fatturazione.algoritmiinterni=negozio.vendita
```

Soluzione 19.n)

Le affermazioni corrette sono le numero 1, e 5. L'affermazione numero 2 è falsa, perché sono le implementazioni a dipendere dal service provider interface. L'affermazione numero 4 è falsa in quanto non è necessario esportare le implementazioni del service provider interface, bensì bisogna usare la direttiva `provides to`.

Soluzione 19.o)

Una possibile soluzione è quella di creare un modulo che chiameremo `com.claudiodesio.rubrica.dati`. All'interno inseriremo le classi `Contatto` e `Dato`, a loro volta modificati per appartenere ad un package che coincide con il nome del modulo:

```
package com.claudiodesio.rubrica.dati;

import java.io.Serializable;

public interface Dato extends Serializable {
}
```

e

```
package com.claudiodesio.rubrica.dati;

import java.io.Serializable;

public class Contatto implements Dato {

    private static final long serialVersionUID = 894240224005652566 1L;

    private String nome;

    private String indirizzo;

    private String numeroDiTelefono;

    public Contatto (String nome, String indirizzo, String numeroDiTelefono) {
        this.nome = nome;
        this.indirizzo = indirizzo;
        this.numeroDiTelefono = numeroDiTelefono;
    }

    public void setNumeroDiTelefono(String numeroDiTelefono) {
        this.numeroDiTelefono = numeroDiTelefono;
    }

    public String getNumeroDiTelefono() {
        return numeroDiTelefono;
    }

    public void setIndirizzo(String indirizzo) {
        this.indirizzo = indirizzo;
    }

    public String getIndirizzo() {
        return indirizzo;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }

    @Override
    public String toString() {
        return "Nome:\t" + nome + "\nIndirizzo:\t" + indirizzo +
          "\nTelefono:\t" + numeroDiTelefono;
    }
}
```

I package delle altre classi li potremo vedere nelle soluzioni dei prossimi esercizi. Il descrittore del modulo sarà il seguente:

```
module com.claudiodesio.rubrica.dati {
    exports com.claudiodesio.rubrica.dati;
}
```

Per compilare il modulo utilizzeremo il seguente comando utilizzando la solita struttura di cartelle che abbiamo utilizzato sinora (l'esercizio completo è nella cartella Codice\capitolo_19\esercizi\19.o del file contenente gli esempi di codice che avete probabilmente scaricato insieme al file che state leggendo, all'indirizzo http://www.claudiodesio.com/java9.html):

```
javac -d mods/com.claudiodesio.rubrica.dati
  src/com.claudiodesio.rubrica.dati/module-info.java
  src/com.claudiodesio.rubrica.dati/com/claudiodesio/rubrica/dati/*.java
```

Soluzione 19.p)

Una possibile soluzione è quella di creare un modulo che chiameremo `com.claudiodesio.rubrica.eccezioni`. All'interno inseriremo le classi `ContattoInesistenteException` e `ContattoEsistenteException`, a loro volta modificate per appartenere ad un package che coincide con il nome del modulo:

```
package com.claudiodesio.rubrica.eccezioni;

import java.io.IOException;

public class ContattoInesistenteException extends IOException {

    private static final long serialVersionUID = 8942402240056525663L;

    public ContattoInesistenteException(String message) {
        super(message);
    }
}
```

e

```
package com.claudiodesio.rubrica.eccezioni;

import java.io.IOException;

public class ContattoEsistenteException extends IOException {

    private static final long serialVersionUID = 8942402240056525662L;
```

```
    public ContattoEsistenteException (String message) {
        super(message);
    }
}
```

Il descrittore del modulo sarà il seguente:

```
module com.claudiodesio.rubrica.eccezioni {
    exports com.claudiodesio.rubrica.eccezioni;
}
```

Per compilare entrambi i moduli definiti, utilizzeremo il seguente comando utilizzando la solita struttura di cartelle che abbiamo utilizzato sinora (l'esercizio completo è nella cartella Codice\capitolo_19\esercizi\19.p del file contenente gli esempi di codice):

```
javac -d mods --module-source-path src
  src/com.claudiodesio.rubrica.dati/module-info.java
  src/com.claudiodesio.rubrica.dati/com/claudiodesio/rubrica/dati/*.java
  src/com.claudiodesio.rubrica.eccezioni/module-info.java
  src/com.claudiodesio.rubrica.eccezioni/com/claudiodesio/rubrica/eccezioni/*.java
```

Soluzione 19.q)

Una possibile soluzione è quella di creare un modulo che chiameremo `com.claudiodesio.rubrica.util`. All'interno abbiamo deciso di inserire le classi `Retriever`, `Executor` e `FileUtils`, a loro volta modificate per appartenere ad un package che coincide con il nome del modulo. Seguono le dichiarazioni delle tre classi:

```
package com.claudiodesio.rubrica.util;

@FunctionalInterface
public interface Retriever<O> {

    O esegui() throws Exception;
}
```

e

```
package com.claudiodesio.rubrica.util;

@FunctionalInterface
public interface Executor {

    void esegui() throws Exception;
}
```

e

```
package com.claudiodesio.rubrica.util;

public class FileUtils {
    public static final String SUFFIX = ".con";

    public static String getNomeFile(String nome) {
        return nome + SUFFIX;
    }
}
```

Il descrittore del modulo sarà il seguente:

```
module com.claudiodesio.rubrica.util {
    exports com.claudiodesio.rubrica.util;
}
```

Per compilare i tre moduli definiti, utilizzeremo il seguente comando utilizzando la solita struttura di cartelle che abbiamo utilizzato sinora (l'esercizio completo è nella cartella Codice\capitolo_19\esercizi\19.q del file contenente gli esempi di codice):

```
javac -d mods --module-source-path src
  src/com.claudiodesio.rubrica.dati/module-info.java
  src/com.claudiodesio.rubrica.dati/com/claudiodesio/rubrica/dati/*.java
  src/com.claudiodesio.rubrica.eccezioni/module-info.java
  src/com.claudiodesio.rubrica.eccezioni/com/claudiodesio/rubrica/
    eccezioni/*.java src/com.claudiodesio.rubrica.util/module-info.java
  src/com.claudiodesio.rubrica.util/com/claudiodesio/rubrica/util/*.java
```

Soluzione 19.r)

La classe che farà da service provider interface è `GestioneSerializzazione`, che faremo appartenere al modulo `com.claudiodesio.rubrica.spi`. Abbiamo modificato la classe per appartenere al package `com.claudiodesio.rubrica.spi`:

```
package com.claudiodesio.rubrica.spi;

import com.claudiodesio.rubrica.dati.Dato;
import java.io.*;
import java.util.*;

public interface GestoreSerializzazione<T extends Dato> {

    void inserisci(T dato) throws IOException;

    T recupera(String id) throws IOException, ClassNotFoundException;
```

```
    void modifica(T dato) throws IOException;

    void rimuovi(String id) throws IOException;
}
```

Il descrittore del modulo sarà il seguente:

```
module com.claudiodesio.rubrica.spi {
    exports com.claudiodesio.rubrica.spi;
    requires com.claudiodesio.rubrica.dati;
}
```

In questo caso c'è stato bisogno di leggere il modulo `com.claudiodesio.rubrica.dati`, visto che la classe `GestoreSerializzazione` utilizza l'interfaccia `Dato`.
Per compilare i quattro moduli definiti sinora utilizzeremo il seguente comando utilizzando la solita struttura di cartelle (l'esercizio completo è nella cartella Codice\capitolo_19\esercizi\19.r del file contenente gli esempi di codice):

```
javac -d mods --module-source-path src
  src/com.claudiodesio.rubrica.dati/module-info.java
  src/com.claudiodesio.rubrica.dati/com/claudiodesio/rubrica/dati/*.java
  src/com.claudiodesio.rubrica.eccezioni/module-info.java
  src/com.claudiodesio.rubrica.eccezioni/com/claudiodesio/rubrica/
    eccezioni/*.java src/com.claudiodesio.rubrica.util/module-info.java
  src/com.claudiodesio.rubrica.util/com/claudiodesio/rubrica/util/*.java
  src/com.claudiodesio.rubrica.spi/module-info.java
  src/com.claudiodesio.rubrica.spi/com/claudiodesio/rubrica/spi/*.java
```

Soluzione 19.s)

Per prima cosa abbiamo dovuto aggiungere alle classi `GestoreFile` e `GestoreFileNIO2`, oltre alla dichiarazione dei package, diversi `import`:

```
package com.claudiodesio.rubrica.io;

import com.claudiodesio.rubrica.spi.GestoreSerializzazione;
import com.claudiodesio.rubrica.dati.Contatto;
import com.claudiodesio.rubrica.eccezioni.*;
import com.claudiodesio.rubrica.util.*;
import java.util.*;
import java.io.*;

public class GestoreFile implements GestoreSerializzazione<Contatto> {

    @Override
    public void inserisci(Contatto contatto)
      throws ContattoEsistenteException, FileNotFoundException,
      IOException {
        Contatto contattoEsistente = getContatto(contatto.getNome());
```

```
        if (contattoEsistente != null) {
            throw new ContattoEsistenteException(
              contatto.getNome() + ": contatto già esistente!");
        }
        registra( contatto);
    }

    @Override
    public Contatto recupera(String nome)
      throws ContattoInesistenteException, ContattoEsistenteException {
        Contatto contatto = getContatto(nome);
        if (contatto == null) {
            throw new ContattoInesistenteException(
              nome + ": contatto non trovato!");
        }
        return contatto;
    }

    @Override
    public void modifica(Contatto contatto)
      throws ContattoInesistenteException, ContattoEsistenteException,
      FileNotFoundException, IOException {
        if (isContattoEsistente(contatto.getNome())) {
            registra(contatto);
        } else {
            throw new ContattoInesistenteException(
              contatto.getNome() + ": contatto non trovato!");
        }
    }

    @Override
    public void rimuovi(String nome) throws ContattoInesistenteException,
      ContattoEsistenteException, FileNotFoundException, IOException {
        File file = new File(FileUtils.getNomeFile(nome));
        if (file.delete()) {
            System.out.println("Contatto " + nome + " cancellato!");
        } else {
            throw new ContattoInesistenteException(
              nome + ": contatto non trovato!");
        }
    }

    private void registra(Contatto contatto)
      throws  FileNotFoundException, IOException {
        try (FileOutputStream fos =
          new FileOutputStream (new File(
            FileUtils.getNomeFile(contatto.getNome())));
        ObjectOutputStream s = new ObjectOutputStream (fos);) {
            s.writeObject (contatto);
            System.out.println("Contatto registrato:\n" + contatto);
        }
    }
```

```
    private boolean isContattoEsistente(String nome) {
        File file = new File(FileUtils.getNomeFile(nome));
        return file.exists();
    }

    private Contatto getContatto(String nome) {
        try (FileInputStream fis = new FileInputStream (
          new File(FileUtils.getNomeFile(nome)));
        ObjectInputStream ois = new ObjectInputStream (fis);) {
            Contatto contatto = (Contatto)ois.readObject();
            System.out.println("Contatto recuperato:\n"+ contatto);
            return contatto;
        } catch (Exception exc) {
            return null;
        }
    }
}
```

e

```
package com.claudiodesio.rubrica.nio;

import com.claudiodesio.rubrica.spi.GestoreSerializzazione;
import com.claudiodesio.rubrica.dati.Contatto;
import com.claudiodesio.rubrica.eccezioni.*;
import com.claudiodesio.rubrica.util.*;
import java.util.*;
import java.io.*;
import java.util.*;
import java.io.*;
import java.nio.file.*;

public class GestoreFileNIO2 implements GestoreSerializzazione<Contatto> {

    @Override
    public void inserisci(Contatto contatto)
      throws ContattoEsistenteException, FileNotFoundException, IOException {
        Path path = Paths.get(FileUtils.getNomeFile(contatto.getNome()));
        if (Files.exists(path)) {
            throw new ContattoEsistenteException(
                contatto.getNome() + ": contatto già esistente!");
        }
        registra(contatto);
    }

    @Override
    public Contatto recupera(String nome)
      throws ContattoInesistenteException, ContattoEsistenteException {
        Contatto contatto = getContatto(nome);
        if (contatto == null) {
```

```
            throw new ContattoInesistenteException(
                nome + ": contatto non trovato!");
        }
        return contatto;
    }

    @Override
    public void modifica(Contatto contatto)
      throws ContattoInesistenteException, ContattoEsistenteException,
      FileNotFoundException, IOException {
        if (isContattoEsistente(contatto.getNome())) {
            registra(contatto);
        } else {
            throw new ContattoInesistenteException(
              contatto.getNome() + ": contatto non trovato!");
        }
    }

    @Override
    public void rimuovi(String nome) throws ContattoInesistenteException,
      ContattoEsistenteException, FileNotFoundException, IOException {
        Path path = Paths.get(FileUtils.getNomeFile(nome));
        if (Files.exists(path)) {
            Files.delete(path);
            System.out.println("Contatto " + nome + " cancellato!");
        } else {
            throw new ContattoInesistenteException(
              nome + ": contatto non trovato!");
        }
    }

    private void registra(Contatto contatto)
      throws FileNotFoundException, IOException {
        Path path = Paths.get(FileUtils.getNomeFile(contatto.getNome()));
        Files.write(path, getBytesDaOggetto(contatto));
        System.out.println("Contatto registrato:\n" + contatto);
    }

    private byte[] getBytesDaOggetto(Object object) throws IOException {
        try (ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutput out = new ObjectOutputStream(bos)) {
            out.writeObject(object);
            return bos.toByteArray();
        }
    }

    private Object getOggettoDaByte(byte[] bytes)
      throws IOException, ClassNotFoundException {
        try (ByteArrayInputStream bis = new ByteArrayInputStream(bytes);
        ObjectInput in = new ObjectInputStream(bis)) {
            return in.readObject();
```

```
        }
    }

    private boolean isContattoEsistente(String nome) {
        Path path = Paths.get(FileUtils.getNomeFile(nome));
        return Files.exists(path);
    }

    private Contatto getContatto(String nome) {
        Path path = Paths.get(FileUtils.getNomeFile(nome));
        byte[] bytes = null;
        Contatto contatto = null;
        try {
            bytes = Files.readAllBytes(path);
            contatto = (Contatto)getOggettoDaByte(bytes);
            System.out.println("Contatto recuperato:\n" + contatto);
        }
        catch (Exception exc) {
            return null;
        }
        return contatto;
    }
}
```

Il primo modulo conterrà `GestoreFile` e sarà chiamato `com.claudiodesio.rubrica.io`, mentre `GestoreFileNIO2` sarà contenuto nel modulo `com.claudiodesio.rubrica.nio`.
Per il primo modulo possiamo creare il seguente descrittore:

```
module com.claudiodesio.rubrica.io {
    //exports com.claudiodesio.rubrica.io;
    provides com.claudiodesio.rubrica.spi.GestoreSerializzazione
      with com.claudiodesio.rubrica.io.GestoreFile;
    requires com.claudiodesio.rubrica.dati;
    requires com.claudiodesio.rubrica.eccezioni;
    requires com.claudiodesio.rubrica.spi;
    requires com.claudiodesio.rubrica.util;
}
```

Mentre per il secondo possiamo creare quest'altro:

```
module com.claudiodesio.rubrica.nio {
    //exports com.claudiodesio.rubrica.nio;
    provides com.claudiodesio.rubrica.spi.GestoreSerializzazione
      with com.claudiodesio.rubrica.nio.GestoreFileNIO2;
    requires com.claudiodesio.rubrica.dati;
    requires com.claudiodesio.rubrica.eccezioni;
    requires com.claudiodesio.rubrica.spi;
    requires com.claudiodesio.rubrica.util;
}
```

Per compilare i sei moduli definiti sinora, utilizzeremo il seguente comando utilizzando la solita struttura di cartelle (l'esercizio completo è nella cartella Codice\capitolo_19\esercizi\19.s del file contenente gli esempi di codice):

```
javac -d mods --module-source-path src
  src/com.claudiodesio.rubrica.dati/module-info.java
  src/com.claudiodesio.rubrica.dati/com/claudiodesio/rubrica/dati/*.java
  src/com.claudiodesio.rubrica.eccezioni/module-info.java
  src/com.claudiodesio.rubrica.eccezioni/com/claudiodesio/rubrica/
    eccezioni/*.java src/com.claudiodesio.rubrica.util/module-info.java
  src/com.claudiodesio.rubrica.util/com/claudiodesio/rubrica/util/*.java
  src/com.claudiodesio.rubrica.spi/module-info.java
  src/com.claudiodesio.rubrica.spi/com/claudiodesio/rubrica/spi/*.java
  src/com.claudiodesio.rubrica.io/module-info.java
  src/com.claudiodesio.rubrica.io/com/claudiodesio/rubrica/io/*.java
  src/com.claudiodesio.rubrica.nio/module-info.java
  src/com.claudiodesio.rubrica.nio/com/claudiodesio/rubrica/nio/*.java
```

Soluzione 19.t)

Modifichiamo la classe `Esercizio18N` in `Esercizio19T` per sfruttare il `ServiceLoader` e caricare i servizi dichiarati nell'esercizio precedente. Inoltre abbiamo rielaborato i diversi `import` e dichiarato il package di appartenenza.

```
package com.claudiodesio.rubrica.test;

import java.util.function.*;
import com.claudiodesio.rubrica.spi.GestoreSerializzazione;
import com.claudiodesio.rubrica.dati.Contatto;
import com.claudiodesio.rubrica.util.*;
import java.util.Iterator;
import java.util.ServiceLoader;

public class Esercizio19T {

    private GestoreSerializzazione<Contatto> gestoreFile;

    private Contatto[] contatti;

    public Esercizio19T(String className) {
        contatti = getContatti();
        gestoreFile = getGestoreSerializzazione(className);
    }

    public GestoreSerializzazione<Contatto>
      getGestoreSerializzazione(String className) {
        ServiceLoader<GestoreSerializzazione> serviceLoader =
          ServiceLoader.load(
```

```
            com.claudiodesio.rubrica.spi.GestoreSerializzazione.class);
        for (GestoreSerializzazione<Contatto> gestoreSerializzazione :
                                                    serviceLoader) {
            if (gestoreSerializzazione.getClass().getSimpleName().
                equals(className)) {
                return gestoreSerializzazione;
            }
        }
        throw new IllegalArgumentException(
          "Nessun gestore di serializzazione trovato per classe = " + className);
    }

    private void eseguiTest() {
        System.out.println("TESTIAMO LA CREAZIONE DEI TRE CONTATTI");
        creaContatti();
        System.out.println("RECUPERIAMO I TRE CONTATTI");
        recuperaContatti();
        System.out.println(
          "TESTIAMO LA CREAZIONE DI UN CONTATTO GIÀ ESISTENTE");
        creaContattoEsistente();
        System.out.println("PROVIAMO A RECUPERARE UN CONTATTO NON ESISTENTE");
        recuperaContattoNonEsistente();
        System.out.println("MODIFICHIAMO UN CONTATTO ESISTENTE");
        modificaContattoEsistente();
        System.out.println("RIMUOVIAMO UN CONTATTO ESISTENTE");
        rimuoviContattoEsistente();
        System.out.println("MODIFICHIAMO UN CONTATTO NON ESISTENTE");
        modificaContattoNonEsistente();
        System.out.println("RIMUOVIAMO UN CONTATTO NON ESISTENTE");
        rimuoviContattoNonEsistente();
    }

    public void creaContattoEsistente() {
        esegui(()->gestoreFile.inserisci(contatti[0]));
    }

    public void modificaContattoEsistente() {
        esegui(()->gestoreFile.modifica(
          new Contatto("Daniele","Via dei microfoni 1","07890")));
    }

    public void rimuoviContattoEsistente() {
        esegui(()->gestoreFile.rimuovi(contatti[2].getNome()));
    }

    public void modificaContattoNonEsistente() {
        esegui(()->gestoreFile.modifica(
          new Contatto("Pluto","Via dei microfoni 1","07890")));
    }
```

```
    public void rimuoviContattoNonEsistente() {
        esegui(()->gestoreFile.rimuovi("Ligeia"));
    }

    public void recuperaContattoNonEsistente() {
        esegui(()->gestoreFile.recupera("Pippo"));
    }

    public void creaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Creazione contatto:\n" + contatto);
            creaContatto(contatto);
        }
    }

    public void recuperaContatti() {
        for (Contatto contatto : contatti) {
            System.out.println("Recupero contatto: " + contatto.getNome());
            recuperaContatto(contatto.getNome());
        }
    }

    public void recuperaContatto(String nomeContatto) {
        esegui(()->gestoreFile.recupera(nomeContatto));
    }

    private void creaContatto(Contatto contatto) {
        esegui(()->gestoreFile.inserisci(contatto));
    }

    private Contatto[] getContatti() {
        Contatto contatto1 =
          new Contatto("Daniele","Via delle chitarre 1","01234560");
        Contatto contatto2 =
          new Contatto("Giovanni","Via delle scienze 2","0565432190");
        Contatto contatto3 =
          new Contatto("Ligeia","Via dei segreti 3","07899921");
        Contatto[] contatti = {contatto1, contatto2, contatto3} ;
        return contatti;
    }

    public <O> O esegui(Retriever<O> retriever) {
        O output = null;
        try {
            output = retriever.esegui();
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
        return output;
    }
```

```
    public void esegui(Executor executor) {
        try {
            executor.esegui();
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public static void main(String args[]) {
        Esercizio19T esercizio19T = new Esercizio19T(args[0]);
        esercizio19T.eseguiTest();
    }
}
```

In grassetto abbiamo evidenziato il metodo factory `getGestoreSerializzazione()` che gestisce quale servizio utilizzare (si basa sulla specifica del nome della classe). Poi abbiamo definito il relativo modulo di cui riportiamo il descrittore di seguito, che può accedere tramite reflection a `GestoreSerializzazione`:

```
module com.claudiodesio.rubrica.test {
    uses com.claudiodesio.rubrica.spi.GestoreSerializzazione;
    requires com.claudiodesio.rubrica.spi;
    requires com.claudiodesio.rubrica.dati;
    requires com.claudiodesio.rubrica.util;
}
```

Possiamo adesso compilare i sette moduli definiti sinora con il seguente comando (come sempre l'esercizio completo è nella cartella Codice\capitolo_19\esercizi\19.t del file contenente gli esempi di codice):

```
javac -d mods --module-source-path src
  src/com.claudiodesio.rubrica.dati/module-info.java
  src/com.claudiodesio.rubrica.dati/com/claudiodesio/rubrica/dati/*.java
  src/com.claudiodesio.rubrica.eccezioni/module-info.java
  src/com.claudiodesio.rubrica.eccezioni/com/claudiodesio/rubrica/
    eccezioni/*.java src/com.claudiodesio.rubrica.util/module-info.java
  src/com.claudiodesio.rubrica.util/com/claudiodesio/rubrica/util/*.java
  src/com.claudiodesio.rubrica.spi/module-info.java
  src/com.claudiodesio.rubrica.spi/com/claudiodesio/rubrica/spi/*.java
  src/com.claudiodesio.rubrica.io/module-info.java
  src/com.claudiodesio.rubrica.io/com/claudiodesio/rubrica/io/*.java
  src/com.claudiodesio.rubrica.nio/module-info.java
  src/com.claudiodesio.rubrica.nio/com/claudiodesio/rubrica/nio/*.java
  src/com.claudiodesio.rubrica.test/module-info.java
  src/com.claudiodesio.rubrica.test/com/claudiodesio/rubrica/test/*.java
```

Per eseguire l'applicazione possiamo utilizzare il seguente comando:

```
java --module-path mods
  -m com.claudiodesio.rubrica.test/com.claudiodesio.rubrica.test.Esercizio19T GestoreFile
```

Che stamperà lo stesso output visto nella soluzione dell'esercizio 18.n e che riportiamo di seguito per comodità:

```
TESTIAMO LA CREAZIONE DEI TRE CONTATTI
Creazione contatto:
Nome:   Daniele
Indirizzo:      Via delle chitarre 1
Telefono:       01234560
Daniele: contatto già esistente!
Creazione contatto:
Nome:   Giovanni
Indirizzo:      Via delle scienze 2
Telefono:       0565432190
Giovanni: contatto già esistente!
Creazione contatto:
Nome:   Ligeia
Indirizzo:      Via dei segreti 3
Telefono:       07899921
Contatto registrato:
Nome:   Ligeia
Indirizzo:      Via dei segreti 3
Telefono:       07899921
RECUPERIAMO I TRE CONTATTI
Recupero contatto: Daniele
Contatto recuperato:
Nome:   Daniele
Indirizzo:      Via dei microfoni 1
Telefono:       07890
Recupero contatto: Giovanni
Contatto recuperato:
Nome:   Giovanni
Indirizzo:      Via delle scienze 2
Telefono:       0565432190
Recupero contatto: Ligeia
Contatto recuperato:
Nome:   Ligeia
Indirizzo:      Via dei segreti 3
Telefono:       07899921
TESTIAMO LA CREAZIONE DI UN CONTATTO GIÀ ESISTENTE
Daniele: contatto già esistente!
PROVIAMO A RECUPERARE UN CONTATTO NON ESISTENTE
Pippo: contatto non trovato!
MODIFICHIAMO UN CONTATTO ESISTENTE
Contatto registrato:
Nome:   Daniele
Indirizzo:      Via dei microfoni 1
Telefono:       07890
```

```
RIMUOVIAMO UN CONTATTO ESISTENTE
Contatto Ligeia cancellato!
MODIFICHIAMO UN CONTATTO NON ESISTENTE
Pluto: contatto non trovato!
RIMUOVIAMO UN CONTATTO NON ESISTENTE
Ligeia: contatto non trovato!
```

Per utilizzare la classe `GestoreFileNIO2` basterà specificarla come argomento da riga di comando al posto di `GestoreFile`.

Soluzione 19.u)

Riscriviamo così il costruttore della classe `Esercizio19U`:

```
public Esercizio19U(String className) {
    contatti = getContatti();
    gestoreFile =
      GestoreSerializzazioneFactory.getGestoreSerializzazione(className);
}
```

nel quale si invoca il metodo (reso statico) `getGestoreSerializzazione()` della classe `GestoreSerializzazioneFactory`. Infatti ecco la classe `GestoreSerializzazioneFactory`:

```
package com.claudiodesio.rubrica.factory;

import com.claudiodesio.rubrica.spi.GestoreSerializzazione;
import com.claudiodesio.rubrica.dati.Contatto;
import java.util.Iterator;
import java.util.ServiceLoader;

public class GestoreSerializzazioneFactory {
    public static GestoreSerializzazione<Contatto>
      getGestoreSerializzazione(String className) {
        ServiceLoader<GestoreSerializzazione> serviceLoader =
          ServiceLoader.load(
          com.claudiodesio.rubrica.spi.GestoreSerializzazione.class);
        for (GestoreSerializzazione gestoreSerializzazione : serviceLoader) {
            if (gestoreSerializzazione.getClass().getSimpleName()
                .equals(className)) {
                return gestoreSerializzazione;
            }
        }
        throw new IllegalArgumentException(
          "Nessun gestore di serializzazione trovato per classe = "
          + className);
    }
}
```

Il descrittore del nuovo modulo `com.claudiodesio.rubrica.factory` è il seguente:

```
module com.claudiodesio.rubrica.factory {
    exports com.claudiodesio.rubrica.factory to com.claudiodesio.rubrica.test;
    requires com.claudiodesio.rubrica.spi;
    requires com.claudiodesio.rubrica.dati;
    uses com.claudiodesio.rubrica.spi.GestoreSerializzazione;
}
```

Mentre il descrittore `com.claudiodesio.rubrica.test` si può semplificare in questo modo:

```
module com.claudiodesio.rubrica.test {
    requires com.claudiodesio.rubrica.spi;
    requires com.claudiodesio.rubrica.factory;
    requires com.claudiodesio.rubrica.dati;
    requires com.claudiodesio.rubrica.util;
}
```

Il comando per compilare tutto sarà:

```
javac -d mods --module-source-path src
  src/com.claudiodesio.rubrica.dati/module-info.java
  src/com.claudiodesio.rubrica.dati/com/claudiodesio/rubrica/dati/*.java
  src/com.claudiodesio.rubrica.eccezioni/module-info.java
  src/com.claudiodesio.rubrica.eccezioni/com/claudiodesio/rubrica/
    eccezioni/*.java src/com.claudiodesio.rubrica.util/module-info.java
  src/com.claudiodesio.rubrica.util/com/claudiodesio/rubrica/util/*.java
  src/com.claudiodesio.rubrica.spi/module-info.java
  src/com.claudiodesio.rubrica.spi/com/claudiodesio/rubrica/spi/*.java
  src/com.claudiodesio.rubrica.io/module-info.java
  src/com.claudiodesio.rubrica.io/com/claudiodesio/rubrica/io/*.java
  src/com.claudiodesio.rubrica.nio/module-info.java
  src/com.claudiodesio.rubrica.nio/com/claudiodesio/rubrica/nio/*.java
  src/com.claudiodesio.rubrica.test/module-info.java
  src/com.claudiodesio.rubrica.test/com/claudiodesio/rubrica/test/*.java
  src/com.claudiodesio.rubrica.factory/com/claudiodesio/rubrica/factory/*.java
  src/com.claudiodesio.rubrica.factory/module-info.java
```

Il comando di esecuzione non cambia rispetto a quell'usato nell'esercizio precedente, così come l'output.

Soluzione 19.v)

Intanto abbiamo rinominato la classe `Esercizio19U` in `Esercizio19V` e ricompilato i moduli. Abbiamo poi creato gli otto file modulari con questo comando:

```
jar --create --file=lib/com.claudiodesio.rubrica.dati.jar
```

```
  --module-version=1.0 -C mods/com.claudiodesio.rubrica.dati .
jar --create --file=lib/com.claudiodesio.rubrica.eccezioni.jar
  --module-version=1.0 -C mods/com.claudiodesio.rubrica.eccezioni .
jar --create --file=lib/com.claudiodesio.rubrica.util.jar
  --module-version=1.0 -C mods/com.claudiodesio.rubrica.util .
jar --create --file=lib/com.claudiodesio.rubrica.spi.jar
  --module-version=1.0 -C mods/com.claudiodesio.rubrica.spi .
jar --create --file=lib/com.claudiodesio.rubrica.io.jar
  --module-version=1.0 -C mods/com.claudiodesio.rubrica.io .
jar --create --file=lib/com.claudiodesio.rubrica.nio.jar
  --module-version=1.0 -C mods/com.claudiodesio.rubrica.nio .
jar --create --file=lib/com.claudiodesio.rubrica.factory.jar
  --module-version=1.0 -C mods/com.claudiodesio.rubrica.factory .
jar --create --file=lib/com.claudiodesio.rubrica.test.jar
  --module-version=1.0
  --main-class=com.claudiodesio.rubrica.test.Esercizio19V
  -C mods/com.claudiodesio.rubrica.test .
```

Poi abbiamo eseguito il modulo con il comando:

```
java -p lib -m com.claudiodesio.rubrica.test GestoreFile
```

o alternativamente con:

```
java -p lib -m com.claudiodesio.rubrica.test GestoreFileNIO2
```

Soluzione 19.z)

Con il seguente comando creiamo il runtime richiesto:

```
jlink --module-path "C:/Program Files/Java/jdk-9.0.1/jmods"
  --add-modules java.base --output javabasert
```

Mentre con il seguente comando eseguiamo il nostro JAR modulare `com.claudiodesio.rubrica.test` con il runtime creato:

```
javabasert\bin\java -p lib -m com.claudiodesio.rubrica.test GestoreFile
```

Come sempre basta sostituire `GestoreFile` con `GestoreFileNIO2` per ottenere la serializzazione nel modo alternativo:

```
javabasert\bin\java -p lib
  -m com.claudiodesio.rubrica.test GestoreFileNIO2
```

Esercizi dell'appendice E

La variabile d'ambiente CLASSPATH

Esercizio E.a)

1. Impacchettare in un file JAR, il package di autenticazione progettato e realizzato nell'esercizio 5.z.

Soluzioni degli esercizi dell'appendice E

Soluzione E.a)

Il comando per impacchettare i file è:

```
jar -cvfm autenticazione.jar manifest.txt
  com/claudiodesio/autenticazione/*.class
```

dove manifest.txt è un file (che deve essere presente nella cartella da dove si sta eseguendo il comando):

```
  Main-Class: com.claudiodesio.autenticazione.Autenticazione

```

Si noti che si deve andare a capo dopo la riga, altrimenti il comando non sarà considerato. Per eseguire l'applicazione utilizzare il seguente comando:

```
java -jar autenticazione.jar
```

Esercizi dell'appendice F

Approfondimento sugli import statici

Esercizio F.a) Static import, Vero o Falso:

1. Gli static import permettono di non referenziare i membri statici importati.
2. Non è possibile dopo avere importato staticamente una variabile, referenziarla all'interno del codice.
3. La seguente importazione non è corretta perché `java.lang` è sempre importato implicitamente:

```
import static java.lang.System.out;
```

4. Non è possibile importare staticamente classi innestate e/o anonime.
5. In alcuni casi gli import statici potrebbero peggiorare la leggibilità dei nostri file.
6. Considerando la seguente enumerazione:

```
package mypackage;
public enum MyEnum {
  A,B,C
}
```

il seguente codice è compilabile correttamente:

```
import static mypackage.MyEnum.*;
public class MyClass {
  public MyClass() {
    out.println(A);
  }
}
```

7. Se utilizziamo gli import statici, si potrebbero importare anche due membri statici con lo stesso nome. Il loro utilizzo all'interno del codice darebbe luogo ad errori in compilazione, se non referenziati.
8. Lo shadowing è un fenomeno che potrebbe verificarsi se si utilizzano gli import statici.
9. Essenzialmente l'utilità degli import statici risiede nella possibilità di scrivere meno codice probabilmente superfluo.
10. Non ha senso importare staticamente una variabile, se poi viene utilizzata una sola volta all'interno del codice.

Soluzioni degli esercizi dell'appendice F

Soluzione F.a) Static import, Vero o Falso:

1. **Vero**.
2. **Falso**.
3. **Falso**.
4. **Falso**.
5. **Vero**.
6. **Falso**, `out` non è importato staticamente.
7. **Vero**.
8. **Vero**.
9. **Vero**.
10. **Vero**.

Esercizi dell'appendice G

Un esempio guidato alla programmazione ad oggetti

Questa appendice andrebbe studiata subito dopo aver studiato il modulo 8. Di conseguenza anche questi esercizi andrebbero fatti dopo aver studiato il polimorfismo. Tuttavia è più facile che il lettore sia in grado di risolverli dopo aver studiato anche altri argomenti.

Per gli esercizi di quest'appendice non vengono presentate soluzioni.

Esercizio G.a)

Riprogettare l'applicazione dell'esempio dell'appendice G, cercando di rispettare le regole dell'Object Orientation.

Raccomandiamo al lettore di cercare una soluzione teorica prima di "buttarsi sul codice". Avere un metodo di approcciare il problema può fare risparmiare ore di debug. Questo metodo dovrà permettere quantomeno di:

1) **Individuare le astrazioni chiave del progetto (le classi più importanti).**
2) **Assegnare loro responsabilità avendo cura dell'astrazione.**
3) **Individuare le relazioni tra esse.**

Utilizzare UML potrebbe essere considerato (anche da chi non l'ha mai utilizzato) un modo per non iniziare a smanettare da subito con il codice.

Non viene presentata soluzione per quest'esercizio in quanto è importante raggiungerne una, qualsiasi essa sia, senza la tentazione di "spiare" quale sia.

Esercizio G.b) Realizzare una semplice applicazione che simuli il funzionamento di una rubrica.

Il lettore si limiti a simulare la seguente situazione: una rubrica contiene informazioni (nome, indirizzo, numero telefonico) su un certo numero di persone (per esempio 5) prestabilito (le informazioni sono pre-introdotte nel metodo `main()`). L'utente dovrà fornire all'applicazione un nome da riga di comando e l'applicazione dovrà restituire le informazioni relative alla persona. Se il nome non è fornito, o se il nome immesso non corrisponde al nome di una persona pre-introdotta dall'applicazione, deve essere restituito un messaggio significativo, non ci sono altri vincoli.
Non è presentata soluzione per quest'esercizio. Anche in questo caso infatti, è importante raggiungerne una, qualsiasi essa sia, senza la tentazione di "spiare" la soluzione.
L'esercizio proposto è presentato spesso a corsi di formazione da me erogati, nella giornata iniziale, per testare il livello della classe. Molto spesso, anche corsisti che si dichiarano "programmatori Java" con esperienza e/o conoscenze, non riescono ad ottenere un risultato accettabile. Ricordiamo una volta di più che questo testo vuole rendere il lettore capace di programmare in Java in modo corretto e senza limiti. Non bisogna avere fretta! Con un po' di pazienza iniziale in più si otterranno risultati sorprendenti. Una volta padroni del linguaggio non esisteranno più ambiguità e misteri, e l'acquisizione di argomenti che oggi sembrano avanzati (EJB, Servlet, etc.) risulterà semplice!

Soluzioni degli esercizi dell'appendice G

Soluzione G.a)

Non è prevista una soluzione per questo esercizio.

Soluzione G.b)

Non è prevista una soluzione per questo esercizio.

Tra gli esercizi del capitolo 18 e 19 viene realizzata una versione modificata di questo esercizio.

Esercizi dell'appendice N

Java e il mondo XML

Esercizio N.a) JAXP, Vero o Falso:

1. Per le specifiche DOM, ogni nodo è equivalente ad un altro e un commento viene visto come un oggetto di tipo `Node`.
2. L'interfaccia `Node` implementa `Text`.
3. Per poter analizzare un documento nella sua interezza con DOM, bisogna utilizzare un metodo ricorsivo.
4. Con il seguente codice:

   ```
   Node n = node.getParentNode().getFirstChild();
   ```

   si raggiunge il primo nodo figlio di `node`.
5. Con il seguente codice:

   ```
   Node n = node.getParentNode().getPreviousSibling();
   ```

   si raggiunge il nodo fratello precedente di `node`.
6. Con il seguente codice:

   ```
   NodeList list = node.getChildNodes();
   Node n = list.item(0);
   ```

   si raggiunge il primo nodo figlio di `node`.

7. Con il seguente codice:

```
Element element = doc.createElement("nuovo");
doc.appendChild(element);
Text text = doc.createTextNode("prova testo");
doc.insertBefore(text, element);
```

viene creato un nodo chiamato `nuovo`, in cui viene aggiunto testo.

8. La rimozione di un nodo provoca la rimozione di tutti i suoi nodi figli.

9. Per analizzare un documento tramite l'interfaccia SAX bisogna estendere la classe `DefaultHandler` ed effettuare l'override dei suoi metodi.

10. Per trasformare un file XML e serializzarlo in un altro file dopo una trasformazione mediante un file XSL, è possibile utilizzare il seguente codice:

```
try {
    TransformerFactory factory = TransformerFactory.newInstance();
    Source source = new StreamSource(new File("input.xml"));
    Result result = new StreamResult(new File("output.xml"));
    Templates template = factory.newTemplates(
      new StreamSource(new FileInputStream("transformer.xsl")));
    Transformer transformer = template.newTransformer();
    transformer.transform(source, result);
} catch (Exception e) {
    e.printStackTrace();
}
```

Soluzioni degli esercizi dell'appendice N

Soluzione N.a) JAXP, Vero o Falso:

1. **Vero**.
2. **Falso**, l'interfaccia `Text` implementa `Node`.
3. **Vero**.
4. **Falso**, si raggiunge il primo nodo fratello di node.
5. **Falso**.
6. **Vero**.
7. **Falso**, il testo viene aggiunto prima del tag con il metodo `insertBefore()`. Sarebbe invece opportuno utilizzare la seguente istruzione per aggiungere il testo all'interno del nuovo tag:

```
element.appendChild(text);
```

8. **Falso**.
9. **Vero**.
10. **Vero**.

Esercizi dell'appendice O

Prima del Framework Collections

Esercizio O.a)

1. Quali delle seguenti affermazioni sono corrette?
2. La classe `HashMap` estende `Hashtable`.
3. La classe `Hashtable` non è thread-safe.
4. La classe `Hashtable` estende `Properties`.
5. `Enumeration` è stata deprecata da quando sono state introdotte le enumerazioni e la parola chiave `enum` in Java 5.
6. L'interfaccia `Enumeration` ha gli stessi metodi dell'interfaccia `Iterator`.

Esercizio O.b)

1. Quali delle seguenti affermazioni sono corrette?
2. La classe `Vector` implementa `List`.
3. La classe `Vector` non è thread-safe.
4. La classe `Vector` non è generica.
5. In generale la classe `Vector`, essendo sincronizzata, offre prestazioni superiori rispetto ad un `ArrayList`.

Soluzioni degli esercizi dell'appendice O

Soluzione O.a)

Nessuna delle affermazioni è corretta!

Soluzione O.b)

Nessuna delle affermazioni è corretta!

Esercizi dell'appendice P

Networking

Esercizio P.a) Networking, Vero o Falso:

1. In una comunicazione di rete devono esistere almeno due socket.
2. Un client, per connettersi ad un server, deve conoscere almeno il suo indirizzo IP e la porta su cui si è posto in ascolto.
3. Un server si può mettere in ascolto anche sulla porta 80, la porta di default dell'HTTP, senza per forza utilizzare quel protocollo. È infatti possibile anche che si comunichi con il protocollo HTTP su una porta diversa dalla 80.
4. Il metodo `accept()` blocca il server in uno stato di "attesa di connessioni". Quando un client si connette, il metodo `accept()` viene eseguito per raccogliere tutte le informazioni del client in un oggetto di tipo `Socket`.
5. Un `ServerSocket` non ha bisogno di dichiarare l'indirizzo IP, ma deve solo dichiarare la porta su cui si metterà in ascolto.

Soluzioni degli esercizi dell'appendice P

Soluzione P.a) Networking, Vero o Falso:

1. **Vero**.
2. **Vero**.
3. **Vero**.
4. **Vero**.
5. **Vero**.

Esercizi dell'appendice Q

Interfacce grafiche (GUI) con AWT, Applet e Swing

Esercizio Q.a) GUI, AWT e Layout Manager, Vero o Falso:

1. Nella progettazione di una GUI è preferibile scegliere soluzioni standard per facilitare l'utilizzo all'utente.
2. Nell'MVC il Model rappresenta i dati, il Controller le operazioni e la View l'interfaccia grafica.
3. Le GUI AWT sono invisibili di default.
4. Per ridefinire l'aspetto grafico di un componente AWT è possibile estenderlo e ridefinire il metodo `paint()`.
5. AWT è basato sul pattern Decorator.
6. In un'applicazione basata su AWT è necessario sempre avere un top level container.
7. È impossibile creare GUI senza layout manager, otterremmo solo eccezioni al runtime.
8. Il `FlowLayout` cambierà la posizione dei suoi componenti in base al ridimensionamento.

9. Il `BorderLayout` cambierà la posizione dei suoi componenti in base al ridimensionamento.

10. Il `GridLayout` cambierà la posizione dei suoi componenti in base al ridimensionamento.

Esercizio Q.b) Gestione degli eventi, Applet e Swing, Vero o Falso:

1. Il modello a delega è basato sul pattern Observer.

2. Senza la registrazione tra la sorgente dell'evento e il gestore dell'evento, l'evento non sarà gestito.

3. Le classi innestate e le classi anonime non sono adatte per implementare gestori di eventi.

4. Una classe innestata può gestire eventi se e solo se è statica.

5. Una classe anonima per essere definita si deve per forza istanziare.

6. Un `ActionListener` può gestire eventi di tipo `MouseListener`.

7. Un pulsante può chiudere una finestra.

8. È possibile (ma non consigliabile) per un gestore di eventi estendere tanti adapter per evitare di scrivere troppo codice.

9. La classe `Applet`, estendendo `Panel`, potrebbe anche essere aggiunta direttamente ad un `Frame`. In tal caso però, i metodi sottoposti a override non verranno chiamati automaticamente.

10. I componenti di Swing (`JComponent`) estendono la classe `Container` di AWT.

Soluzioni degli esercizi dell'appendice Q

Soluzione Q.a) GUI, AWT e Layout Manager, Vero o Falso:

1. **Vero**.
2. **Falso**, in particolare il Model rappresenta l'intera applicazione composta da dati e funzionalità.
3. **Vero**.
4. **Vero**.
5. **Falso**, è basata sul pattern Composite che, nonostante abbia alcuni punti di contatto con il Decorator, è completamente diverso.
6. **Vero**.
7. **Falso**, ma perderemmo la robustezza e la consistenza della GUI.
8. **Vero**.
9. **Falso**.
10. **Falso**.

Soluzione Q.b) Gestione degli eventi, Applet e Swing, Vero o Falso:

1. **Vero**.

2. **Vero**.
3. **Falso**.
4. **Falso**.
5. **Vero**.
6. **Falso**, solo di tipo `ActionListener`.
7. **Vero**, può sfruttare il metodo `System.exit(0)`, ma non è pertinente con gli eventi di tipo `WindowEvent`.
8. **Vero**.
9. **Vero**.
10. **Vero**.

Esercizi dell'appendice R

Java Database Connectivity

Esercizio R.a) JDBC, Vero o Falso:

1. `Connection` è solo un'interfaccia.
2. Un'applicazione JDBC è indipendente dal database solo se si parametrizzano le stringhe relative al driver, lo URL di connessione, lo username e la password.
3. Se si inoltra ad un particolare database un comando non standard SQL 2, questo comando funzionerà solo su quel database. In questo modo si perde l'indipendenza dal database, a meno di controlli o parametrizzazioni.
4. Per eliminare un record bisogna utilizzare il metodo `executeQuery()`.
5. Per aggiornare un record bisogna utilizzare il metodo `executeUpdate()`.
6. `CallableStatement` è una sottointerfaccia di `PreparedStatement`. `PreparedStatement` è una sottointerfaccia di `Statement`.
7. Per eseguire una stored procedure bisogna utilizzare il metodo `execute()`.
8. L'autocommit è impostato a `true` per default.
9. In ambienti enterprise dove si eseguono applicazioni in contesti Java EE, `DataSource` va utilizzato in luogo di `Driver`.
10. Per poter modificare il risultato di una query, è necessario utilizzare un oggetto `RowSet` in luogo di un `ResultSet`.

Esercizio R.b)

Riprendere l'esercizio G.d dell'appendice G (oppure si può partire anche dalla soluzione dell'esercizio 18.n, o anche provare a rendere il tutto modulare studiando la soluzione dell'esercizio 19.v), dove viene richiesto di creare una rubrica. Di seguito viene riproposta la traccia per comodità:

Realizzare un'applicazione che simuli il funzionamento di una rubrica.

Il lettore si limiti a simulare la seguente situazione: una rubrica contiene informazioni (nome, indirizzo, numero telefonico) su un certo numero di persone (per esempio 5) prestabilito (le informazioni sono pre-introdotte nel metodo).

L'utente dovrà fornire all'applicazione un nome da riga di comando e l'applicazione dovrà restituire le informazioni relative alla persona. Se il nome non è fornito, o se il nome immesso non corrisponde al nome di una persona pre-introdotta dall'applicazione, deve essere restituito un messaggio significativo. Il lettore non ha altri vincoli.

Realizzare la parte di persistenza dei dati usando un database come Apache Derby, dopo aver creato una base dati relazionale.

Ricordiamo che non è presentata soluzione per quest'esercizio.

Soluzioni degli esercizi dell'appendice R

Soluzione R.a) JDBC, Vero o Falso:

1. **Vero**.
2. **Vero**.
3. **Vero**.
4. **Falso**.
5. **Vero**.
6. **Vero**.
7. **Vero**.
8. **Vero**.
9. **Vero**.
10. **Falso**.

Soluzione R.b)

Non è presentata una soluzione per questo esercizio.

Esercizi dell'appendice S

Interfacce grafiche: introduzione a JavaFX

Esercizio S.a) GUI, AWT e Layout Manager, Vero o Falso:

1. È possibile creare interfacce miste che usano librerie di Swing e di JavaFX.
2. Per visualizzare un'interfaccia grafica bisogna utilizzare un oggetto `Stage` posizionato su un oggetto `Scene`.
3. Le interfacce create con JavaFX si creano usando layout pane, o quei pannelli che definiscono come i componenti aggiunti su di essi devono essere posizionati.
4. Il `GridPane` permette di posizionare i suoi componenti in una griglia le cui celle devono essere della stessa dimensione.
5. Il linguaggio FXML permette di usare gli stessi componenti grafici che utilizza JavaFX in maniera dichiarativa.
6. È possibile usare un file CSS solo con interfacce create con FXML.
7. JavaFX, diversamente da Swing/AWT, non gestisce gli eventi con un modello a delega. Infatti si usano le espressioni labda e non i gestori degli eventi.
8. Il binding permette a due oggetti che definiscono delle property JavaFX di legarsi e aggiornare il loro stato in base all'aggiornamento dell'altro oggetto.

9. Gli effetti speciali definiti JavaFX che permettono il movimento, terminano con il suffisso *Motion*, infatti estendono la classe astratta `Motion`.

10. È possibile eseguire un'applicazione JavaFX come applet.

Esercizio S.b)

Riprendere l'esercizio R.b dove è stata creata una rubrica che si basa su un database relazionale. Di seguito viene riproposta la traccia per comodità:
Realizzare un'applicazione che simuli il funzionamento di una rubrica.
Il lettore si limiti a simulare la seguente situazione: una rubrica contiene informazioni (nome, indirizzo, numero telefonico) su un certo numero di persone (per esempio 5) prestabilito (le informazioni sono pre-introdotte nel metodo `main()`). L'utente dovrà fornire all'applicazione un nome da riga di comando e l'applicazione dovrà restituire le informazioni relative alla persona. Se il nome non è fornito, o se il nome immesso non corrisponde al nome di una persona pre-introdotta dall'applicazione, deve essere restituito un messaggio significativo. Il lettore non ha altri vincoli. Realizzare la parte di persistenza dei dati usando un database come Apache Derby, dopo aver creato una base dati relazionale.
Realizzare un'interfaccia grafica a piacere con Java FX, per rendere il programma completo.

Ricordiamo che non è presentata soluzione per quest'esercizio, e che, se non si è studiata ancora la parte relativa ai database, è possibile partire dalla soluzione dell'esercizio 18.n, o anche provare a rendere il tutto modulare studiando la soluzione dell'esercizio 19.v.

Soluzioni degli esercizi dell'appendice S

Soluzione S.a) GUI, AWT e Layout Manager, Vero o Falso:

1. **Vero**.
2. **Falso**, è esattamente il contrario.
3. **Vero**.
4. **Falso**, tali caratteristiche sono del `TilePane`.
5. **Vero**.
6. **Falso**.
7. **Falso**.
8. **Vero**.
9. **Falso**, abbiamo visto un esempio di sfocatura tramite la classe `BlurMotion` che però non implementa movimenti ed estende la classe astratta `Effect`. Invece le classi che implementano un movimento estendono `Transition` e di conseguenza terminano con la parola `Transition`.
10. **Vero**.

Soluzione S.b)

Non è prevista una soluzione per questo esercizio.