Corso di Metodologie di Programmazione
Corso di Laurea in Informatica
Sapienza Università di Roma

Esercizi di preparazione all'esame
Prof. Roberto Navigli

1 Parte 1

1.1 Heap & Stack

Esercizio 1.1.1. Disegnare lo stato della memoria (heap e stack) appena prima del termine dell'esecuzione del metodo `main` della classe seguente:

```
public class MyClass<T>
{
   private T x;
   private int y;

   public MyClass(T x, int y) { this.x = x; this.y = y; }

   public T getX() { return x; }
   public int getY() { return y; }

   public static void main(String[] args)
   {
      MyClass<Integer> a = new MyClass<Integer>(5, 2);
      MyClass<String> b = new MyClass<String>("stringa", 42);
      MyClass<Integer> g = a;

      String s;

      // fotografa qui lo stato della memoria
   }
}
```

Esercizio 1.1.2. Disegnare lo stato della memoria (heap e stack) appena prima del termine dell'esecuzione del metodo `main` della classe seguente:

```
public class Tornello
{
  static private int passaggi;

  public void passa() { passaggi++; }
  public static int getPassaggi() { return passaggi; }

  public static void main(String[] args)
  {
    Tornello t1 = new Tornello();
    t1.passa();
    Tornello t2 = new Tornello();
    for (int k = 0; k < 10; k++) t2.passa();
    int g;
    String s = null;
    // fotografa qui lo stato della memoria
  }
}
```

Esercizio 1.1.3. Disegnare lo stato della memoria (heap e stack) appena prima del termine dell'esecuzione del metodo `main` della classe seguente:

```
public class Utente
{
  static private int utenti;
  private String nome;

  public Utente(String nome) { this.nome = nome; utenti++; }
  public String getUtente() { return nome; }

  public static void main(String[] args)
  {
    int k = 0;
    String s = "mario";
    Utente u1 = new Utente(s);
    Utente u2 = new Utente("luigi");
    k++;
    // fotografa qui lo stato della memoria
  }
}
```

Esercizio 1.1.4. Disegnare lo stato della memoria (heap e stack) appena prima del termine dell'esecuzione del metodo `main` della classe `Poesia`:

```
public class Poesia
{
  private String[] versi;
  private int k;

  public Poesia(int length) { versi = new String[length]; }
  public void addVerso(String verso) { versi[k++] = verso; }
  public int getLength() { return versi.length; }

  public static void main(String[] args)
  {
    String millumino = "M'illumino";
    Poesia p = new Poesia(3);
    p.addVerso(millumino);
    p.addVerso("d'immenso");
    int length = p.getLength();
    // fotografa qui lo stato della memoria
  }
}
```

1.2 Identifica e correggi gli errori (1)

Esercizio 1.2.1. Identificare (cerchiandoli) e giustificare (a margine) gli errori nella seguente classe:

```
public class Errori
{
  private char k;

  public Errori(int k)
  {
    this.k = k;
  }

  public long getK() { return k }

  public static void main(String[] args)
  {
    k++;
    Errori e = new Errori();
    getK();
  }
}
```

Esercizio 1.2.2. Identificare (cerchiandoli) e giustificare (a margine) gli errori nella seguente classe:

```
public class PuntoIntero
{
  private int x;
  private int y;

  public PuntoIntero(String x)
  {
    String[] coppia = x.split(",");
    x = (int)coppia[0];
    y = (int)coppia[1];
  }

  public toString() { return x+','+y; }

  static public int getX() { return x; }
  static public int getY() { return y; }
}
```

Esercizio 1.2.3. Identificare (cerchiandoli) e giustificare (a margine) gli errori nella seguente classe:

```
public class MyString
{
  private String s;

  public MyString(String s)
  {
    MyString.s = s;
```

```
  }

  public static void main(String[] args)
  {
    if (args.length > 0)
    {
      MyString s = new MyString("ciao");
      MyString t = new MyString(args[0]);
      if (s == t) System.out.println("I due oggetti contengono la stessa stringa");
    }

    s = "hello";
  }
}
```

Esercizio 1.2.4. Identificare (cerchiandoli) e giustificare (a margine) gli errori nella seguente classe:

```
public class Punto
{
  private int x;
  private int y;

  public Punto(long x, long y)
  {
    this.x = x;
    this.y = y;
  }

  public String toString()
  {
    return "("+x+","+y+")";
  }

  private void dummy()
  {
    long z = x;
  }

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

1.3 Array e Stringhe

Esercizio 1.3.1. Progettare una classe `GestoreArray` costruita a partire da un array di interi. La classe implementa i seguenti metodi:

- `indexOf` che, dato in input un intero, restituisce la posizione dell'intero all'interno dell'array, se presente, `-1` altrimenti.
- `concat` che, preso in input un altro array di interi, lo concatena in fondo all'array dell'oggetto. Il metodo non restituisce nulla.
- `concatNoDup` che, preso in input un altro array di interi, concatena in fondo all'array dell'oggetto solo quegli elementi dell'array in input che non sono già presenti nell'oggetto. Il metodo restituisce il numero di elementi scartati. Ad esempio, `new GestoreArray(new int[] { 1, 2 }).concatNoDup(new int[] { 6, 2, 3 });` memorizza all'interno dell' oggetto l'array { 1, 2, 6, 3 } perché `2` era già presente nell'array iniziale, restituendo `1`.
- `replace` che, dati in input due interi `x` e `y`, sostituisce la prima occorrenza di `x` con il valore di `y` all'interno dell'array. Se l'occorrenza viene sostituita, il metodo restituisce `true`, altrimenti `false`.

Per la gestione dell'espansione dell'array **non è consentito** l'uso dei metodi della classe `Arrays`.

Esercizio 1.3.2. Progettare la classe `StringaMangiona`. Un oggetto della classe viene costruito passando in ingresso una stringa di tipo `String` (ad esempio, `new StringaMangiona("aiuola")`). La classe implementa i seguenti metodi:

- `toString`: restituisce la stringa;
- `length`: restituisce la lunghezza della stringa;
- `getCarattere`: prende in input una posizione k e restituisce in output il carattere della stringa nella posizione k;
- `mangiaLettera`: modifica la stringa dell'oggetto eliminando tutte le occorrenze della lettera passata in input (es. `new StringaMangiona("aiuola").mangiaLettera('a')` modifica la stringa interna dell'oggetto in "`iuol`");
- `slurp`: mangia tutte le lettere della stringa interna dell'oggetto;
- `mangiaStringaMangiona`: data in input un'altra `StringaMangiona s`, mangia tutti i propri caratteri contenuti anche nella stringa mangiona `s` (ad esempio, l'esecuzione di:

```
new StringaMangiona("aiuola").mangiaStringaMangiona(new StringaMangiona("ala"));
```

modifica la stringa interna del primo oggetto in "`iuo`").

- `getTotaleMangiate`: restituisce la somma delle occorrenze di lettere mangiate finora dall'oggetto (ad esempio, a seguito dell'esecuzione di:

```
StringaMangiona s = new StringaMangiona("aabcbbb");
s.mangiaLettera('a');
s.mangiaLettera('b');
s.slurp();
```

  `getTotaleMangiate()` restituirà 7).

Esercizio 1.3.3. Progettare la classe `EstrazioneDelLotto`. La classe implementa i seguenti metodi:

- `estrai`: estrae, memorizzandoli, 5 numeri tra 1 e 90 (non importa se con o senza ripetizioni);
- `toString`: restituisce l'ultima estrazione sotto forma di stringa (ad esempio, "`42, 11, 78, 90, 12`");
- `numeriContenuti`: prende in input una giocata (sequenza fino a 10 numeri) e restituisce quanti dei numeri giocati sono anche contenuti nell'ultima estrazione (ad esempio, 2 indica un ambo, 3 un terno, ecc.);
- `vincita`: prende in input una giocata e restituisce un booleano che indica se c'è stata una vincita (dall'ambo in poi);
- `getTotaleNumeriEstratti`: restituisce la somma totale dei numeri finora estratti (ad esempio, dopo le estrazioni "`42, 11, 78, 90, 12`" e "`14, 77, 65, 32, 51`", restituisce `472`).

Esercizio 1.3.4. Progettare una classe `Calcolatrice` i cui oggetti rappresentano una calcolatrice con memoria espandibile. Ogni oggetto calcolatrice è posto a un valore iniziale pari a `0.0` e possiede una memoria inizialmente vuota. La classe implementa i seguenti metodi:

- `somma` che, dato in input un valore, lo somma al valore memorizzato nella calcolatrice, aggiornandone il valore.
- `memorizza` che memorizza l'attuale valore della calcolatrice nella prossima cella di memoria della calcolatrice.
- `azzera` che pone a `0.0` il valore della calcolatrice.
- `recupera` che, dato in input un intero `k`, recupera dalla memoria il valore associato alla cella di memoria `k`, se essa esiste, e lo imposta come attuale valore della calcolatrice, restituendo `true`. Se la cella di memoria non è mai stata usata il metodo restituisce `false`.

- `toString` che restituisce una stringa che esprime la sequenza delle somme effettuate.

Ad esempio, il seguente codice:

```
Calcolatrice c = new Calcolatrice();
c.somma(5); c.somma(3);
System.out.println(c.toString());
c.memorizza(); c.azzera(); c.recupera(0); c.somma(1);
System.out.println(c.toString());
c.memorizza(); c.azzera(); c.recupera(1);
System.out.println(c.toString());
```

stampa prima `0+5+3`, poi `8+1` e poi `9`.

2 Parte 2

2.1 Ereditarietà e polimorfismo

Esercizio 2.1.1. Si vuole progettare un piccolo videogioco *shoot'em up* spaziale. Gli elementi del videogioco sono i seguenti:

- La navicella del giocatore, un oggetto della classe `Navicella`, dotata di una posizione (x, y) sul tabellone (inizialmente pari a $(0, 2)$) e di uno stato `laser` che determina se la navicella sta sparando oppure no. La navicella è dotata dei seguenti metodi:
  - un metodo `toString()` che ne restituisce la rappresentazione "N";
  - due metodi `su` e `giu` che incrementano la posizione y.
  - metodi setter/getter per lo stato dello sparo (attivo o non attivo);
- I nemici, che possono essere di due tipi: `Alieno` e `Meteorite`, entrambi anch'essi dotati di una posizione (x, y), dotati di un metodo `toString` che ne restituisce l'iniziale ("A" o "M") e di un metodo `prossimoPasso` il cui comportamento consiste, nel caso dell'alieno, nel decrementare di 1 la posizione x dell'alieno, mentre nel caso del meteorite, nell'incrementare di 1 la posizione y del meteorite e decrementare di 1 la posizione x (il meteorite si sposta verso sinistra e verso il basso).

Infine si progetti una classe `Sparatutto` i cui oggetti rappresentano istanze del videogioco. Ogni oggetto della classe contiene un tabellone 5x10 in cui in ogni cella si trova un elemento dello sfondo. Esistono due possibili tipi di sfondo: uno sfondo vuoto (la cui rappresentazione sotto forma di stringa è un carattere di spazio) e uno sfondo stella (la cui rappresentazione è il carattere asterisco). Si utilizzino le classi per modellare gli sfondi.

Ogni oggetto di `Sparatutto` viene inizializzato con un tabellone vuoto cui vengono aggiunte 3 stelle in posizione casuale (x, y). L'oggetto viene costruito con una navicella e una lista di nemici. La classe è inoltre dotata dei metodi:

- `toString` che restituisce il tabellone 5x10 con le celle di sfondo laddove in tale posizione non si trovi un giocatore o un nemico e la rappresentazione a stringa del giocatore o del nemico altrimenti. Se il giocatore ha lo sparo laser attivo, una striscia orizzontale di trattini viene stampata a partire dalla posizione della navicella fino alla fine del tabellone. Un esempio di stringa restituita in caso di sparo laser attivo è il seguente:

```
  *    A  \n
   M  *   \n
N---------\n
   *      \n
      A   \n
```

- `prossimoPasso` che richiama il metodo `prossimoPasso` di ciascun nemico sul tabellone.

Esercizio 2.1.2. Si vuole progettare una classe `GiocoDelPacMan` le cui istanze sono costruite creando un tabellone NxN di celle, ciascuna delle quali può contenere una `CellaMuro` oppure una `CellaLibera`. Il tabellone, dato il valore N iniziale in input, è costruito con celle tutte di tipo `CellaLibera`. Ogni istanza della classe `GiocoDelPacMan` possiede un riferimento al giocatore `PacMan`, creato in fase di costruzione con posizione $(0, 0)$, e una lista di istanze della classe `Fantasmino`, popolata con 2 fantasmini in fase di costruzione dell'oggetto. I fantasmini vengono generati da un apposito metodo di comodo della classe `GiocoDelPacMan` che ne stabilisce una posizione casuale ma non sovrapposta.

La classe è dotata dei seguenti metodi:

- `impostaMuro` che, date in ingresso una posizione x e y, impostino un muro in quella posizione del tabellone. Si prevedano eventuali eccezioni.

- `toString` che mostra lo stato del tabellone in quell'istante, stampando per ogni cella: la rappresentazione a stringa delle celle ("M" nel caso di muro, " " oppure "." nel caso di una cella libera, secondo se possiede o non possiede la pallina) laddove queste siano libere da fantasmini o da PacMan; la rappresentazione del personaggio altrimenti. In fondo alla stringa viene anche inserito il punteggio attuale del giocatore `PacMan` (si veda sotto). Ad esempio:

```
MMMMMMMMM\n
M..P.....M\n
M.MMM.MM.M\n
M..F..F..M\n
MMMMMMMMM\n
\n
Punti: 0\n
```

- `esegui` che muove PacMan e i fantasmini chiamandone i corrispondenti metodi `sposta`.

- `termina` che termina il gioco, chiamando il metodo `System.exit(0);`

La classe `CellaLibera`, inizialmente costruita con lo stato "pallina presente", possiede degli appositi metodi getter e setter per leggere e modificare tale stato.

Le classi `PacMan` e `Fantasmino` costruiscono gli oggetti a partire da un'istanza del `GiocoDelPacMan` e da una posizione x e y e possiedono un metodo `sposta` che ne modifica la posizione x e y in una direzione casuale ma valida (ovvero una cella contigua a quella in cui si trova il personaggio, non protetta da muro). Se la cella contiene una pallina, la pallina viene "mangiata" (ovvero eliminata dalla cella) e il punteggio di PacMan viene incrementato di 1. Se la cella contiene un fantasmino, il gioco termina. La classe `PacMan` è dotata anche del metodo `getPunteggio` che ne restituisce il punteggio attuale.

Esercizio 2.1.3. Si vuole progettare una classe `Treno` le cui istanze sono costruite a partire da un vagone di tipo `Locomotiva`. Il treno è dotato dei seguenti metodi:

- `aggiungiVagone`, che aggiunge in coda al treno il vagone fornito in input;
- `dividiTreno` che, data una posizione `k` e una locomotiva in input, toglie dal treno attuale tutti i vagoni dalla posizione `k` in poi e li inserisce in un nuovo treno, accodandoli alla locomotiva in input. Gestire gli opportuni casi di eccezione.
- `toString`, che restituisce una stringa contenente i nomi dei vagoni del treno separati da "`--`", da sinistra verso destra. Il primo vagone sarà quindi quello della locomotiva. Un esempio è la stringa:

```
"Locomotiva--VagonePasseggeri--VagoneLetto--VagoneMerci"
```

Ogni vagone contiene le seguenti informazioni: destinazione e numero di posti disponibili. Inoltre ogni vagone dispone di un metodo `getNumeroPostiLiberi` per ottenere il numero di posti liberi rimasti nel vagone.

Si vogliono progettare, oltre alla `Locomotiva`, diversi tipi di `Vagone`, ovvero:

- `VagonePasseggeri`, che possiede 50 posti;
- `VagoneLetto`, che possiede 10 posti;
- `VagoneMerci`, che non possiede posti liberi.

Codificare le costanti in modo appropriato. Tutti i vagoni dispongono del metodo `occupa` che, dato in ingresso un passeggero, lo registrano nell'elenco dei passeggeri del vagone se questo dispone di posti liberi, emettendo un'eccezione altrimenti. Al momento della registrazione, viene stampato a video il messaggio: "`Passeggero <nome> registrato`", dove `<nome>` è il nome del passeggero.

Mentre il `VagonePasseggeri` ammette tutti i tipi di passeggeri, il `VagoneLetto` ammette solo passeggeri assonnati.

Infine si progettino i due tipi di passeggero `Passeggero` (costruita a partire da un nome del passeggero) e `PasseggeroAssonnato` (costruita con un nome standard "passeggero assonnato"). Entrambe implementano il metodo `getNome` che ne restituisce il nome.

Esercizio 2.1.4. Si vogliono progettare diversi tipi di filtro, le cui istanze sono costruite senza parametri oppure a partire da una lista di interi. Ogni tipo di filtro è dotato dei seguenti metodi:

- `filtra` che opera una qualche operazione di filtro sulla lista di interi dell'oggetto, restituendo una nuova lista filtrata. Il metodo emette eccezione nel caso in cui la lista non sia stata fornita in input al momento della costruzione.
- una seconda versione di `filtra` che opera la medesima operazione di filtro, ma su una lista fornita in input.

Si progettino i seguenti tipo di `Filtro`:

- `FiltroIntero`, la cui operazione `filtra` restituisce una nuova lista in cui tutte le occorrenze di un determinato intero nella lista vengono scartate; tale intero viene specificato al momento della costruzione del filtro.
- `FiltroPrimo`, la cui operazione di filtro consiste nell'eliminazione nella nuova lista di tutte le occorrenze del primo intero nella lista.
- `FiltroDispari`, la cui operazione di filtro consiste nell'eliminazione nella nuova lista di tutte le occorrenze in posizione dispari nella lista.
- `MultiFiltro`, i cui oggetti vengono costruiti con una lista di filtri e la cui operazione `filtra` applica i filtri in sequenza (ovvero l'output di un filtro è fornito in input al successivo, ecc.).

Esercizio 2.1.5. L'obiettivo di questo esercizio è quello di progettare una rete sociale. Le due principali componenti sono le classi `ReteSociale` e `Utente`.

Progettare la classe `Utente` che mantiene le seguenti informazioni: nome e cognome dell'utente, lista di amici, richieste di amicizia pendenti. Inoltre la classe prevede un meccanismo di visibilità delle amicizie agli altri utenti: l'elenco dei propri amici può essere reso visibile a tutti o solo ai propri amici.

La classe prevede le seguenti operazioni:

- `richiediAmicizia`: permette a un utente di richiedere l'amicizia di un altro utente;
- `accettaAmicizia`: permette di accettare la richiesta di amicizia pendente di un altro utente;

- `getRichiesteDiAmicizia`: restituisce l'elenco delle richieste di amicizia;
- `getAmiciDi`: restituisce l'elenco degli amici dell'utente `u` fornito in input se la visibilità lo permette (ovvero se le amicizie di `u` sono visibili a tutti oppure se l'utente su cui si esegue il metodo è amico di `u`). Per esempio, se Orlando è amico di Olimpia e Astolfo, egli può in ogni caso ottenerne la lista di amici; al contrario, non essendo amico di Medoro (con visibilità degli amici a tutti) e Angelica (con visibilità degli amici solo ai propri amici), Orlando potrà visualizzare solo gli amici di Medoro.

Prevedere le opportune eccezioni `AmiciziaNonRichiestaException` e `AmiciNonVisibiliException`.

Progettare inoltre i seguenti tipi di utenti:

- `SuperUtente`: è in grado di ottenere gli amici di qualsiasi utente, indipendentemente dalla visibilità da questi impostata;
- `UtenteHacker`: fornisce un metodo aggiuntivo `hackera` che svuota l'elenco degli amici di un utente fornito in ingresso;
- `UtenteFastidioso`: nel richiedere l'amicizia di un altro utente, effettua la richiesta 3 volte di seguito;

Progettare infine una classe `ReteSociale` che rappresenti una rete di amicizie. La classe contiene l'elenco (inizialmente vuoto) di utenti della rete. Permette inoltre di aggiungere nuovi utenti (`aggiungiUtente`) e di ottenere l'elenco di utenti aventi un determinato cognome (`getUtentiPerCognome`).

Esercizio 2.1.6. L'obiettivo di questo esercizio è quello di modellare una pagina nel linguaggio di markup HTML in forma semplificata, come illustrato di seguito. Ad esempio, si vuole rappresentare la seguente pagina:

```
<html><p><b>bella</b> pe tutti</p></html>
```

Si noti che una pagina HTML ha una struttura ricorsiva in cui ogni elemento può essere costituito da uno o più sottoelementi. Nell'esempio mostrato sopra, il tag `<html>` contiene il tag `<p>` che a sua volta contiene una sequenza di due elementi: un tag `<b>` e un elemento di testo `'' pe tutti''`. Ogni tag deve essere aperto e chiuso come nell'esempio sopra riportato.

Progettare una gerarchia di classi che modelli la seguente rappresentazione di elementi HTML. Un `Elemento` può essere di tre tipi, ovvero `Tag`, `SequenzaDiElementi` ed `ElementoTesto`:

1. Il tipo `Tag` fornisce le seguenti operazioni comuni:
    - un costruttore comune a tutti i tag: un tag è costruito passando in input l'`Elemento` in esso contenuto;

- `getTagName`: restituisce il nome del tag;
- `getElementoInterno`: restituisce l'elemento contenuto all'interno del tag;
- `toString`: restituisce il codice HTML del tag (inclusa la rappresentazione ricorsiva del tag interno).

Progettare le seguenti specializzazioni del tipo `Tag`: `TagHTML` (rappresenta il tag `<html>`), `TagParagrafo` (rappresenta il tag `<p>`), `TagGrassetto` (rappresenta il tag `<b>`).

2. Il tipo `SequenzaDiElementi` è costruito a partire da una sequenza di oggetti di tipo `Elemento` e sovrascrive il metodo `toString`, il quale restituisce la concatenazione della rappresentazione sotto forma di stringa dei singoli elementi della sequenza.

3. Il tipo `ElementoTesto` rappresenta una stringa di testo e la restituisce mediante il metodo `toString`.

Progettare una classe `PaginaHTML` che rappresenti una pagina nel linguaggio di markup HTML in forma semplificata costruita a partire da un tag `<html>` (ovvero un'istanza della classe `TagHTML`). La classe sovrascrive il metodo `toString` in modo da restituire la rappresentazione HTML della stessa. Ad esempio, la pagina HTML riportata all'inizio dell'esercizio può essere costruita e visualizzata come segue:

```
PaginaHTML p = new PaginaHTML(
 new TagHTML(
  new TagParagrafo(
   new SequenzaDiElementi(
     new TagGrassetto(new ElementoTesto("bella")), new ElementoTesto(" pe tutti")
   )
  )
 )
);
System.out.println(p);
```

Esercizio 2.1.7. L'obiettivo di questo esercizio è quello di modellare una piccola macchina virtuale che esegua una sequenza di istruzioni in un linguaggio assembly semplificato. Ad esempio, si vuole rappresentare ed eseguire il seguente codice assembly:

```
MOV EAX, 3
MOV EBX, 5
MOV ECX, 8
ADD EAX, EBX
CMP EAX, ECX
CALL MY_PROC
```

dove `EAX`, `EBX` ed `ECX` sono registri della macchina virtuale, `MOV`, `ADD`, `CMP` sono istruzioni binarie, come specificato nel seguito. Un'istruzione binaria prende in ingresso due operandi, uno di tipo registro e l'altro di tipo registro o intero. Infine `CALL` è un'istruzione unaria che prende in ingresso una procedura.

In dettaglio, progettare le quattro istruzioni previste dalla macchina virtuale dotate del metodo `esegui`:

- `Mov`, il cui metodo `esegui` copia il valore del secondo operando all'interno del registro passato come primo operando;
- `Add`, il cui metodo `esegui` modifica il valore contenuto nel registro passato come primo operando sommandogli il valore del secondo operando;
- `Cmp`, il cui metodo `esegui` confronta i valori dei due operandi e memorizza all'interno del registro passato come primo operando: 0 se i due valori sono uguali, -1 se il primo valore è minore del secondo, +1 altrimenti;
- `Call`, il cui metodo `esegui` esegue le istruzioni di una `Procedura`, ovvero una classe costruita a partire da una lista di altre istruzioni e anch'essa dotata del metodo `esegui`.

Progettare inoltre la classe `Procedura`, `Operando` e le sue specializzazioni `Registro` (dotato di un nome) e `Intero`.

Progettare infine una classe `MyVirtualMachine` che fornisca un metodo `esegui` il quale, data una `Procedura`, ne esegue le istruzioni in sequenza. Ad esempio, una virtual machine può essere costruita come segue (`Arrays.asList` è un metodo statico che converte un array in una lista):

```
Registro eax = new Registro("EAX");
Registro ebx = new Registro("EBX");
Registro ecx = new Registro("ECX");
Registro edx = new Registro("EDX");

Istruzione[] codice1 = new Istruzione[] {
```

```
      new Mov(edx, new Intero(0)),
      new Add(edx, ecx) };
Proc proc1 = new Proc(Arrays.asList(codice1));

Istruzione[] codice2 = new Istruzione[] {
      new Mov(eax, new Intero(3)),
      new Mov(ebx, new Intero(5)),
      new Mov(ecx, new Intero(8)),
      new Add(eax, ebx),
      new Cmp(eax, ecx),
      new Call(proc1),
};
Proc main_proc = new Proc(Arrays.asList(codice2));

MyVirtualMachine vm = new MyVirtualMachine();
vm.esegui(main_proc);
```

Esercizio 2.1.8. Progettare un sistema ferroviario costituito da treni e binari, come specificato nel seguito.

Un `Treno` possiede una velocità e contiene l'informazione sullo stato delle porte (aperte o chiuse). Inoltre fornisce i seguenti metodi comuni a tutti i tipi di treno:

- `frena`: frena il treno azzerandone la velocità;
- `entraInStazione`: frena il treno;
- `setPorte`: dato in ingresso un booleano, imposta le porte allo stato corrispondente (aperte o chiuse).

Progettare inoltre le classi `Interregionale` e `TAV`. Quest'ultima ridefinisce il metodo `entraInStazione` il quale, oltre a frenare, apre automaticamente le porte del treno.

Un generico `Binario` è costruito a partire dal binario successivo da percorrere e fornisce il metodo `getSuccessivo`, che restituisce il prossimo binario da percorrere. Inoltre tutti i binari hanno in comune un metodo `percorri` che, dato in input un `Treno`, effettua eventuali operazioni (che dipendono dal tipo di binario, come spiegato sotto) e restituisce in output il prossimo binario che il treno deve percorrere.

Progettare i seguenti tipi di binario:

- `BinarioSemplice`: implementa il metodo `percorri` semplicemente restituendo il prossimo binario;
- `BinarioStazione`: implementa il metodo `percorri` facendo entrare il treno in stazione e restituendo il prossimo binario da percorrere;

- `BinarioSemaforo`: viene costruito fornendo, oltre al binario successivo da percorrere, anche la durata in millisecondi di uno stato del semaforo (rosso o verde). Quest'ultima viene utilizzata in un secondo metodo `rosso()`, per determinare lo stato del semaforo (suggerimento: sfruttare opportunamente il resto della divisione tra `System.currentTimeMillis()` – l'attuale orario in millisecondi – e `durataMillisecondi`). Infine la classe ridefinisce il metodo `percorri` restituendo il prossimo binario se il treno è ad alta velocità (TAV) o il semaforo è verde. Altrimenti il metodo restituisce il binario attuale (facendo rimanere il treno fermo).

- `BinarioScambio`: viene costruito fornendo in input due binari successivi da percorrere. Inizialmente il primo dei due binari è selezionato come binario successivo. La classe implementa un metodo `scambia` che permette di selezionare, rendendolo percorribile, il binario attualmente non selezionato (come in un vero scambio ferroviario). Il metodo `percorri` restituirà quindi tale binario quale prossimo binario che il treno dovrà percorrere.

Non è richiesto l'uso di eccezioni. Il seguente codice crea una sequenza di binari e la fa percorrere a un treno interregionale:

```
Treno t = new Interregionale(50);
Binario b = new BinarioSemplice(
              new BinarioScambio(
                new BinarioSemaforo(
                  new BinarioSemplice(null), 30000),
                new BinarioStazione(null)));

while(b != null)
{
  // prossimo binario
  b = b.percorri(t);
}
```

2.2 Identifica e correggi gli errori (2)

Esercizio 2.2.1. Identificare (cerchiandoli) e giustificare (a margine) gli errori nella seguente classe:

```
abstract public class Classe1
{
   public int print();
}

abstract public class Classe2
{
   private Integer k;
}

public class Classe3 extends Classe1, Classe2
{
   public void print()
   {
      if (new java.util.Random().nextBoolean()) throw new Exception();
   }
}
```

Esercizio 2.2.2. Identificare (cerchiandoli) e giustificare (a margine) gli errori nella seguente classe:

```
public class Classe1
{
   private Integer s;

   abstract public int nextInt(int k);
   public Classe1(Integer s) { this.s = s; }
   public Classe1() { this(0.0); }
}

public class Classe2 extends Classe1
{
   @Override
   public int nextInt(int k) throws Exception
   {
      if (k < 0) throw new Exception();
      return k+1;
   }
}
```

Esercizio 2.2.3. Identificare (cerchiandoli) e giustificare (a margine) gli errori nella seguente classe:

```
abstract public class Classe1
{
   private double x;

   public Classe1() { }
   public Classe1(double x) { this.x = x; }
   public void print() { System.out.println("A"); }
```

```
}

public class Classe2 extends Classe1
{
   public Classe2() { super(5.0); }

   @Override
   public void print(int k)
   {
      System.out.println(k);
   }
   public Classe1 getX() { return new Classe1(); }
}
```

Esercizio 2.2.4. Identificare (cerchiandoli) e giustificare (a margine) gli errori nella seguente classe:

```
abstract public class Classe1
{
   private int k;

   public Classe1(double k) { this.k = 0; }
   public Classe1(int k) { this.k = k; this(5.0); }
   public void print() { System.out.println("Classe1"); }
}

public class Classe2 extends Classe1
{
   public void print()
   {
      System.out.println("Classe2");
      return k;
   }
   public void error()
   {
      if (new java.util.Random().nextInt() < 5) throw new Exception();
   }
}
```

Esercizio 2.2.5. Identificare (cerchiandoli) e giustificare (a margine) gli errori nella seguente classe:

```
abstract public class B
{
   public int printB();
}

abstract public class C
{
   private String s;
}

public class D extends B, C
{
   public void printD()
   {
```

```
      if (new java.util.Random().nextBoolean()) throw new Exception();
   }
}
```

Esercizio 2.2.6. Identificare (cerchiandoli) e giustificare (a margine) gli errori nella seguente classe:

```
public class C
{
   private String s;

   abstract public int nextInt(int k);
   public C(String s) { this.s = s; }
   public C() { this(0); }
}

public class D extends C
{
   @Override
   public int nextInt(int k) throws Exception
   {
      if (k < 0) throw new Exception();
      return k+1;
   }
}
```

Esercizio 2.2.7. Identificare (cerchiandoli) e giustificare (a margine) gli errori nella seguente classe:

```
abstract public class A
{
   private double x;

   public A() { }
   public A(double x) { this.x = x; }
   public void print() { System.out.println("A"); }
   public void tumDeDum()
   {
      int[] a = new int[5];
      for (int j = 0; j <= 5; j++) System.out.println(a[j]);
   }
}

public class B extends A
{
   public int print()
   {
      System.out.println("B");
      return 0;
   }
   public double getX() { return x; }
}
```

Esercizio 2.2.8. Identificare (cerchiandoli) e giustificare (a margine) gli errori nella seguente classe:

```
abstract public class A
{
   private int k;

   public A(int k) { this.k = k; }
   public void print() { System.out.println("A"); }
}

public class B extends A
{
   public void print()
   {
      System.out.println("B");
      return k;
   }
   public void error()
   {
      if (new java.util.Random().nextBoolean()) throw new Exception();
   }
}
```

2.3 Interfacce

Esercizio 2.3.1. Progettare un'interfaccia `Trimmable` che espone un metodo `trim` per "ripulire" gli estremi di un oggetto. Progettare quindi due classi che implementino la suddetta interfaccia:

- `TrimmableString`, i cui oggetti sono costruiti con una stringa. Il metodo `trim` elimina gli spazi (anche ripetuti) all'inizio e alla fine della stringa (ad esempio, `new TrimmableString(" ciao ").trim()` imposta la stringa dell'oggetto a "`ciao`").
- `TrimmableIntList`, costruito con una lista di interi, il cui metodo `trim` modifica la lista eliminando eventuali valori `0` (anche ripetuti) all'inizio e alla fine della lista.

Si prevedano esplicitamente eventuali casi di eccezione.

Esercizio 2.3.2. Progettare un'interfaccia `Accendibile` che espone un metodo `accendi` e un metodo `spegni` i quali modificano lo stato dell'oggetto in questione. Progettare quindi due classi che implementino la suddetta interfaccia:

- `Dispositivo`, i cui oggetti sono costruiti con un messaggio di benvenuto e un messaggio di saluto. All'accensione viene stampato il messaggio di benvenuto, allo spegnimento il messaggio di saluto.
- `Telecomando`, costruito con un oggetto di tipo `Dispositivo`, che permette di accendere o spegnere il dispositivo in questione.

Si prevedano esplicitamente eventuali casi di eccezione.

Esercizio 2.3.3. Progettare un'interfaccia `Cercabile` che espone un metodo `cerca` il quale cerca un carattere all'interno di una sequenza, restituendo un booleano relativo all'esito della ricerca. Progettare quindi due classi che implementino la suddetta interfaccia:

- `StringaCercabile`, i cui oggetti sono costruiti a partire da una stringa, il cui metodo `cerca` restituisce `true` se il carattere fornito in input è contenuto nella stringa.
- `ListaCercabileDiInteri`, i cui oggetti contengono una lista inizialmente vuota, che implementa un metodo `aggiungi` il quale, dato un intero, aggiunge l'intero alla lista e il metodo `cerca` che, dato il carattere in input, restituisce l'esito della ricerca del valore del carattere all'interno della lista.

Si prevedano esplicitamente eventuali casi di eccezione.

Esercizio 2.3.4. Progettare un'interfaccia `SaliScendi` che espone un metodo `sali` e un metodo `scendi` i quali modificano la posizione dell'oggetto in questione. Progettare quindi due classi che implementino la suddetta interfaccia:

- `AstaBandiera`, costruita con un oggetto di tipo `Bandiera`, che permette di far salire o scendere la bandiera. `AstaBandiera` è dotata di un metodo `getBandiera()` che restituisce il riferimento alla bandiera se essa è alzata, `null` altrimenti.
- `Ascensore`, costruito con il numero di piani dell'edificio e inizialmente posizionato al piano 0, che permette di far salire e scendere l'ascensore di un piano a ogni chiamata dei metodi dell'interfaccia.

Non è richiesta l'implementazione della classe `Bandiera`. Si prevedano esplicitamente eventuali casi di eccezione.

Esercizio 2.3.5. Progettare un'interfaccia `Resettabile` che espone un metodo `reset` il quale ripristina un oggetto al suo stato originario di costruzione. Progettare quindi due classi che implementano la suddetta interfaccia:

- `Contatore`, che espone un metodo `conta` il quale incrementa un contatore inizialmente pari a 0. Ad esempio, a seguito delle istruzioni:

  ```
  Contatore c = new Contatore(); for (int k = 0; k < 10; k++) c.conta(); c.reset();
  ```

  il contatore sarà pari a 0.
- `Punto`, costruita con tre `double` `x`, `y` e `z` e con i corrispondenti metodi di `get` e `set`. Ad esempio, a seguito delle istruzioni `Punto p = new Punto(10.5, 0.5, 0.1); p.setX(7.0); p.reset();` il valore di `p.x` sarà `10.5`.

Esercizio 2.3.6. Progettare un'interfaccia `Totalizzabile` che espone un metodo `getTotale` il quale restituisce un intero pari al totale degli elementi di un oggetto. Progettare quindi due classi che implementano la suddetta interfaccia:

- `SequenzaDiInteri`, i cui oggetti sono costruiti con una lista di interi. Il metodo `getTotale` restituisce la somma degli interi nella lista.
- `Frase`, i cui oggetti sono costruiti con un array di stringhe. Il metodo `getTotale` restituisce come totale la somma delle rappresentazioni intere delle stringhe nell'array (ad esempio, `new Frase(``123'', ``42'').getTotale()` restituisce `165`).

Si prevedano esplicitamente eventuali casi di eccezione.

Esercizio 2.3.7. Progettare un'interfaccia `ScambiaCoppie` che espone un metodo `scambia` il quale modifica l'ordine di una sequenza scambiando gli elementi adiacenti a due a due. Progettare quindi due classi che implementano la suddetta interfaccia:

- `Stringa`, costruita con una stringa. Il metodo `scambia` modifica la stringa scambiandone i caratteri a coppie (ad esempio, `new Stringa(''ciao'').scambia()` modifica la stringa in `''icoa''`.
- `SequenzaDiInteri`, i cui oggetti sono costruiti con una sequenza di interi. Il metodo `scambia` inverte gli interi a coppie (ad esempio, `new SequenzaDiInteri(2, 4, 5, 9).scambia()` modifica la sequenza in `4, 2, 9, 5`).

Prevedere il caso d'eccezione in cui nessuno scambio sia possibile (se la sequenza ha lunghezza ≤ 1).

Esercizio 2.3.8. Progettare un'interfaccia `Aggiungibile` che espone un metodo `aggiungi` il quale aggiunge un oggetto di tipo `Object` a una sequenza nella posizione più naturale per essa. Analogamente progettare un'interfaccia `Estraibile` che espone un metodo `estrai` il quale estrae il primo oggetto nella sequenza (eliminandolo da essa). Progettare quindi due classi che implementano la suddetta interfaccia:

- `UfficioPostale`, inizialmente vuota, che permette di aggiungere in coda un nuovo `Cliente` ed estrarre il primo cliente in coda.
- `PilaDiPiatti`, inizialmente vuota, che permette di aggiungere un nuovo `Piatto` in cima alla pila ed estrarre il piatto in cima alla pila.

Non è richiesta l'implementazione del cliente e del piatto. Si prevedano esplicitamente eventuali casi di eccezione.

2.4 Interfacce Funzionali ed Espressioni Lambda

Nei seguenti esercizi, si assuma sempre che gli argomenti in input non sono pari a `null`.

Esercizio 2.4.1. Si definisca in una riga una variabile del tipo di un'interfaccia funzionale standard del package `java.util.function` e le si associ una lambda (**non** un riferimento a metodo) che, dato in input un intero, ne calcola il quadrato. Si scriva quindi una seconda riga per invocare il metodo dell'interfaccia.

Esercizio 2.4.2. Si definisca in una riga una variabile del tipo di un'interfaccia funzionale standard del package `java.util.function` e le si associ una lambda (**non** un riferimento a metodo) che, data in input una stringa, la stampi a video. Si scriva quindi una seconda riga per invocare il metodo dell'interfaccia.

Esercizio 2.4.3. Si definisca in una riga una variabile del tipo di un'interfaccia funzionale standard del package `java.util.function` e le si associ una lambda (**non** un riferimento a metodo) che, senza prendendo in input una stringa e un intero, restituisca un booleano che verifica se la stringa è della lunghezza pari all'intero fornito in input. Si scriva quindi una seconda riga per invocare il metodo dell'interfaccia.

Esercizio 2.4.4. Si definisca in una riga una variabile del tipo di un'interfaccia funzionale standard del package `java.util.function` e le si associ una lambda (**non** un riferimento a metodo) che, senza prendere nulla in input, restituisca un numero intero casuale. Si scriva quindi una seconda riga per invocare il metodo dell'interfaccia.

Esercizio 2.4.5. Si definisca in una riga una variabile del tipo di un'interfaccia funzionale standard del package `java.util.function` e le si associ una lambda (**non** un riferimento a metodo) che, prendendo in input una stringa, restituisca un booleano che verifica se la stringa è vuota oppure no. Si scriva quindi una seconda riga per invocare il metodo dell'interfaccia.

Esercizio 2.4.6. Progettare un'interfaccia funzionale `ElaboraStringhe` che esponga un metodo `elabora` il quale, data in input una stringa, restituisca un'altra stringa. Scrivere quindi le seguenti espressioni lambda da assegnare alla riga `ElaboraStringhe e =` in modo tale che:

- l'espressione restituisca la rappresentazione sotto forma di stringa della lunghezza della stringa in input;
- l'espressione restituisca i primi 5 caratteri della stringa o, se più piccola, la stringa per intero.

Esercizio 2.4.7. Progettare un'interfaccia funzionale `VerificaStringhe` che esponga un metodo `verifica` il quale, date in input due stringhe `s1` e `s2`, restituisca un booleano. Scrivere quindi le seguenti espressioni lambda da assegnare alla riga `VerificaStringhe v =` in modo tale che:

- l'espressione restituisca `true` se `s2` è contenuto in `s1`, `false` altrimenti;
- l'espressione restituisca `true` se la lunghezza di `s1` è maggiore di quella di `s2` ed `s1` non contiene `s2` come suffisso.

Esercizio 2.4.8. Progettare un'interfaccia funzionale `FunzioneSuInsieme` che esponga un metodo `applica` il quale, dato in input un insieme di interi e un intero k, restituisca un intero. Scrivere quindi le seguenti espressioni lambda da assegnare alla riga `FunzioneSuInsieme f =` in modo tale che:

- l'espressione restituisca la somma dei valori $\leq k$ nell'insieme;
- l'espressione restituisca il minimo valore nell'insieme; `null` se l'insieme è vuoto.

Esercizio 2.4.9. Progettare un'interfaccia funzionale `FunzioneSuListe` che esponga un metodo `applica` il quale, data in input una lista di stringhe e una stringa `s`, restituisca un intero. Scrivere quindi le seguenti espressioni lambda da assegnare alla riga `FunzioneSuListe f =` in modo tale che:

- l'espressione restituisca il numero di stringhe nella lista che iniziano con la stringa `s`;
- l'espressione restituisca la posizione del primo elemento nella lista che contiene la stringa `s`, -1 se non `s` è mai contenuto.

Esercizio 2.4.10. Progettare un'interfaccia funzionale `Funzione` che esponga un metodo `applica` il quale, data in input una lista di interi e un intero `k`, restituisca un insieme di interi. Scrivere quindi le seguenti espressioni lambda da assegnare alla riga `Funzione f =` in modo tale che:

- l'espressione restituisca l'insieme dei primi k elementi della lista;
- l'espressione restituisca un insieme contenente i valori della lista in input che sono $\leq k$.

Esercizio 2.4.11. Progettare un'interfaccia funzionale `FunzioneStringaIntero` che esponga un metodo `applica` il quale, dati in input una stringa e un intero `k`, restituisca una stringa. Scrivere quindi le seguenti espressioni lambda da assegnare alla riga `FunzioneStringaIntero f =` in modo tale che:

- l'espressione restituisca una stringa che è pari alla ripetizione `k` volte della stringa fornita in input;

- l'espressione restituisca gli ultimi `k` caratteri della stringa in input o la stringa per intero se la sua dimensione è $\leq k$.

3 Parte 3

3.1 Interfacce notevoli

Esercizio 3.1.1. Si modifichi la seguente classe in modo da renderla comparabile secondo l'ordinamento naturale degli interi, apportando le modifiche **direttamente sul foglio**.

```
public class MyArray
{
   private int a[];

   public MyArray(int... interi) { a = interi; }
}
```

Bonus: Si mostri il codice (una riga per caso) per creare un insieme contenente `MyArray a = new MyArray(new int[] { 1, 2, 3 })` e `MyArray b = new MyArray(new int[] { 2, 1 })` ordinato secondo: a) il criterio di ordinamento lessicografico specificato sopra, b) secondo il criterio di inserimento e c) secondo un criterio apparentemente casuale. Si usino direttamente i riferimenti `a` e `b`.

Esercizio 3.1.2. Si modifichi la seguente classe in modo da renderla iterabile apportando le modifiche **direttamente sul foglio**. Si faccia in modo che l'iteratore implementi anche il metodo `reset` (mancante nell'interfaccia standard) mediante una specializzazione dell'interfaccia.

```
public class MyArray
{
   private String a[];

   public MyArray(String... a) { this.a = a; }
}
```

Bonus: Modificare ulteriormente la classe in modo che due istanze con medesimo contenuto siano considerate uguali (ad esempio, l'inserimento di due istanze `new MyArray(new String[] { "abc", "de" })` e `new MyArray(new String[] { "abc", "de" })` in un `HashSet` causerà l'inserimento solo del primo array, essendo la seconda istanza uguale alla prima).

Esercizio 3.1.3. Si modifichi la seguente classe apportando le modifiche **direttamente sul foglio** in modo da rendere le istanze della classe comparabili in ordine lessicografico.

```
public class Coppia
{
   private int x;
   private int y;

   public Coppia(int x, int y) { this.x = x; this.y = y; }
}
```

Bonus: Modificare ulteriormente la classe in modo che due coppie con medesimo contenuto siano sempre considerate uguali (ad esempio, l'inserimento di coppie uguali, quali `new Coppia(0, 42)` e `new Coppia(0, 42)`, in un `HashSet` causerà l'inserimento solo della prima coppia, essendo la seconda uguale). Perché questo comportamento non era garantito dalle modifiche effettuate prima?

Esercizio 3.1.4. Si modifichi la seguente classe in modo da renderla iterabile apportando le modifiche **direttamente sul foglio**.

```
public class MyList
{
   private Elemento first;
   private class Elemento
   {
      private int val;
      private Elemento next;
      public Elemento(int val, Elemento next) { this.val = val; this.next = next; }
   }

   public MyList(int... interi) { for (int k : interi) first = new Elemento(k, first); }
}
```

Bonus: Modificare ulteriormente la classe in modo che due liste con medesimo contenuto siano considerate uguali (ad esempio, l'inserimento delle liste `new MyList(1, 2, 42)` e `new MyList(1, 2, 42)` in un `HashSet` causerà l'inserimento solo della prima lista, essendo la seconda uguale alla prima).

Esercizio 3.1.5. Si consideri la seguente classe:

```
public class StringList
{
    private String lista = "";

    public StringList(String... str)
    {
        for (String s : str) lista += s+"\t";
    }
}
```

La classe memorizza una lista di stringhe sotto forma di un'unica stringa che separa le singole stringhe mediante il carattere tab '\t'. Modificare la classe in modo che i suoi oggetti permettano l'iterazione sulle singole stringhe. Ad esempio:

```
for (String s : new StringList("a", "b", "c")) System.out.println(s);
```

stamperà:

```
a
b
c
```

Esercizio 3.1.6. Questo esercizio è una variante dell'esercizio precedente: scrivere una classe `MyString`, i cui oggetti sono costruiti con una stringa, iterabile sui caratteri della stringa stessa. Ad esempio:

```
for (char c : new MyString("ciao")) System.out.println(c);
```

stamperà:

```
c
i
a
o
```

Esercizio 3.1.7. Modificare la classe `MyString` dell'esercizio precedente in modo che le sue istanze siano confrontabili per lunghezza. La classe ridefinisce anche il metodo `toString` per restituire la stringa contenuta. Ad esempio:

```
TreeSet<String> set = new TreeSet<>(Arrays.asList(new MyString("abc"), new MyString("10"), new MyString("a")))
System.out.println(set);
```

stamperà:

```
[a, 10, abc]
```

3.2 Generici e Collection

Esercizio 3.2.1. Si definisca una classe generica `MinMax` che contiene due campi, `min` e `max`, del tipo generico.

Scrivere quindi un metodo generico (appartenente a un'altra classe che non è necessario specificare) che, data in input una collection di elementi, restituisce un oggetto di tipo `MinMax` contenente il minimo e il massimo tra gli oggetti della collection fornita in input.

Ad esempio, la seguente chiamata `getMinMax(Arrays.asList("ciao", "aaa", "bb", "aabb", "zzz")` restituisce un oggetto di tipo `MinMax<String>` contenente "`aaa`" come minimo e "`zzz`" come massimo.

Esercizio 3.2.2. Progettare una classe generica i cui oggetti rappresentino una collezione di coppie `(x, y)` senza ripetizioni. La classe dispone dei seguenti metodi:

- `aggiungi`: data in input una coppia (`x`, `y`), la aggiunge alla collezione.

- `elimina`: data in input una coppia (`x`, `y`), la elimina se è presente.
- `getCoppiePerX`: data la coordinata `x`, fornisce l'insieme delle coppie aventi la coordinata x specificata;
- `toString`: restituisce una rappresentazione a stringa della collezione di coppie analoga a quella del file fornito in ingresso, in cui i valori `x` delle coppie sono ordinati secondo l'ordinamento naturale del tipo di `x`.

Esercizio 3.2.3. Si definisca una classe **generica** per gestire multimappe. Una multimappa mantiene associazioni (chiave, insieme di valori) tali che ad ogni chiave è associato un insieme di valori. Il tipo delle chiavi può essere diverso da quello dei valori.

Oltre a definire un costruttore che crea una multimappa vuota, la classe deve definire i seguenti metodi:

- Un metodo `put` che, presa in input una chiave e un valore, aggiunge l'associazione alla multimappa, restituendo `true` se il valore non era già contenuto nell'insieme associato alla chiave, `false` altrimenti. Il metodo gestisce la situazione in cui la chiave specificata non esista, creando la nuova associazione.
- Un metodo `get` che, presa in input una chiave, se la chiave è presente restituisce l'insieme ad essa associato, altrimenti restituisce `null`.
- Un metodo `contains` che, presa in input una chiave e un valore, restituisce `true` se l'associazione tra la chiave e il valore è contenuta nella multimappa, `false` altrimenti.
- Un metodo `intersect` che, presa in input una chiave `k` e un insieme di valori `set`, se la chiave è presente rende l'insieme dei valori associato alla chiave uguale all'intersezione tra l'insieme originale e l'insieme `set` preso in input. Se `set` è pari a `null`, la chiave `k` viene rimossa dalla multimappa. Se la chiave non è presente, il metodo lancia l'eccezione `IllegalArgumentException`.
- Un metodo `intersectMultiMappa` che, presa in input una multimappa dello stesso tipo, rende la multimappa dell'oggetto su cui il metodo è invocato uguale all'intersezione delle due multimappe. In altre parole, rimarranno in questa multimappa solamente le chiavi che sono presenti anche nella multimappa presa in input e, per ognuna di queste chiavi, l'insieme dei valori diventerà uguale all'intersezione degli insiemi dei valori associati alla chiave nelle due multimappe.

Esercizio 3.2.4. Si definisca una classe **generica** per gestire multiinsiemi. Un multiinsieme è un insieme che può contenere più copie dello stesso elemento (ad esempio, { 1, 1, 2, 3, 3 }).

Oltre a definire un costruttore che crea una multiinsieme vuoto, la classe deve definire i seguenti metodi:

- Un metodo `add` che, preso in input un valore, lo aggiunge al multiinsieme, restituendo `true` se il valore non era già contenuto nel multiinsieme, `false` altrimenti.
- Un metodo `get` che, preso in input un valore, restituisce il numero di copie di tale valore nel multinsieme (0 se il valore non è contenuto).
- Un metodo `contains` che, preso in input un valore, restituisce `true` se il valore è contenuto nel multiinsieme, `false` altrimenti.
- Un metodo `toSet` che restituisce l'insieme dei valori contenuti nel multiinsieme, ovvero senza duplicati.
- Un metodo `intersect` che, preso in input un multiinsieme `set` dello stesso tipo dell'oggetto su cui il metodo è invocato, modifica il multiinsieme di quest'ultimo in modo da contenere l'intersezione tra se stesso e `set`. Ad esempio, dato il multinsieme { 1, 1, 1, 2, 4, 4, 5 }, l'intersezione con il multiinsieme { 1, 1, 2, 4, 7, 7 } modifica il primo in { 1, 1, 2, 4 }.

Esercizio 3.2.5. Si definisca una classe generica `GString` i cui oggetti sono costruiti con una stringa. La stringa permette la cosiddetta interpolazione, ovvero la possibilità di specificare il nome di una variabile preceduto dal carattere $, da sostituire in seguito con il valore ad essa associato. La classe espone il metodo `put` che, preso in input il nome di una variabile e il valore associato del tipo generico, salva la corrispondenza all'interno dell'oggetto, da utilizzare in seguito per l'interpolazione.

Il metodo `toString`, opportunamente ridefinito, restituisce la stringa in cui sono stati sostituiti i valori correnti di ciascuna variabile in essa specificata. Ad esempio:

```
GString<Integer> s = new GString("Due interi: $x e $y");
s.put("x", 42);
s.put("y", 0);
System.out.println(s);
```

stamperà:

```
Due interi: 42 e 0
```

Esercizio 3.2.6. Si implementi un metodo generico `freq` che, presa una lista di oggetti in input, stampi secondo il loro ordine naturale stabilito gli oggetti contenuti nella lista insieme alla frequenza con cui ciascun oggetto appare. Ad esempio, data la seguente chiamata:

```
freq(Arrays.asList("oh", "capitano", "mio", "capitano"));
```

il metodo visualizza:

```
capitano 2
mio 1
oh 1
```

Che modifica dovremmo apportare al metodo per stampare gli oggetti in un ordine non definito?

3.3 Ricorsione

Esercizio 3.3.1. Scrivere un metodo **ricorsivo** `sommaFinoA` che, dato in input un intero `n`, restituisce la somma di tutti i valori da 1 a n. Ad esempio, `sommaFinoA(5)` restituisce `15`.

Esplicitare chiaramente nel codice il o i casi base e il passo ricorsivo.

Esercizio 3.3.2. Scrivere un metodo **ricorsivo** `min` che, dato un array di interi, restituisca il minimo elemento dell'array. Ad esempio, `min(new int[] { 5, 2, -1, 42 })` restituisce `-1`.

Esplicitare chiaramente nel codice il o i casi base e il passo ricorsivo.

Esercizio 3.3.3. Scrivere un metodo **ricorsivo** `inverti` che, data in input una stringa, restituisca la stringa invertita. Ad esempio, `inverti("roma")` restituisce `"amor"`.

Esplicitare chiaramente nel codice il o i casi base e il passo ricorsivo.

Esercizio 3.3.4. Scrivere un metodo **ricorsivo** `potenza` che, dati in input due interi x e y, restituisca la potenza x^y. Ad esempio, `potenza(2, 5)` restituisce `32`.

Esplicitare chiaramente nel codice il o i casi base e il passo ricorsivo.

Esercizio 3.3.5. Scrivere un metodo **ricorsivo** `scartaParentesi` che, data una stringa, ne restituisca un'altra che contenga solo il testo fuori delle parentesi. Ad esempio, `scartaParentesi("costante (42)((eo)--)e variabile(abc)")` restituisce "`costante e variabile`". Si gestiscano eventuali eccezioni **(+2 punti)**.

Esplicitare chiaramente nel codice il o i casi base e il passo ricorsivo.

Esercizio 3.3.6. Una stringa binaria generalizzata è una stringa costituita da una sequenza di cifre `0`, `1` e una cifra "jolly" `*`. Il simbolo `*` indica la possibilità per quella cifra di assumere valore `0` oppure `1`. Ad esempio, la stringa binaria generalizzata "`10*1*`" può essere espansa nei numeri "`10010`", "`10011`", "`10110`", "`10111`" sostituendo tutti i possibili valori al posto dell'`*`.

Implementare un metodo **ricorsivo** `espandi` che, data in input una stringa binaria generalizzata, stampi a video tutte le possibili stringhe binarie ottenute sostituendo `0` oppure `1` al simbolo `*`.

Esplicitare chiaramente nel codice il o i casi base e il passo ricorsivo.

Esercizio 3.3.7. Un array di interi può essere interpretato come un insieme di posizioni. Ad esempio, definendo `int[] a = new int[] { 3, 2, -1, 1, 2 }` possiamo seguire un percorso partendo da `a[0]` e spostandoci quindi nella posizione dell'array specificata da `a[0]`, ovvero `3`, spostandoci quindi nella posizione specificata in `a[3]`, ovvero `1`, spostandoci quindi nella posizione specificata in `a[1]`, ovvero `2`. Il percorso si interrompe quando la cella dell'array contiene `-1`, come avviene in `a[2]`. Come risultato otteniamo un "percorso", che nell'esempio è `0, 3, 1, 2, -1`.

Scrivere un metodo **ricorsivo** `percorri` che, dato un array di interi, parta dalla posizione `0` dell'array e restituisca il percorso seguito come illustrato sopra.

Esercizio 3.3.8. Implementare un metodo **ricorsivo** `cambiaBase` che, dato in input un intero e una base ≤ 10, ne restituisca la rappresentazione stringa nella base specificata. Ad esempio, `cambiaBase(12, 2)` restituisce "`1100`", mentre `cambiaBase(12, 8)` restituisce "`14`".

Si ricorda che il cambiamento di base si ottiene mediante la concatenazione dei resti delle divisioni successive del numero iniziale per la base. Ad esempio, $12/2 = 6$ con resto **0**, $6/2 = 3$ con resto **0**, $3/2 = 1$ con resto **1**, $1/2 = 0$ con resto **1**, per cui il numero risultante è, concatenando i resti dall'ultimo al primo, "`1100`".

Esplicitare chiaramente nel codice il o i casi base e il passo ricorsivo.

Esercizio 3.3.9. Scrivere un metodo ricorsivo `genera` che, dato in input un insieme di caratteri e un intero `k`, restituisca l'insieme di tutte le possibili stringhe di lunghezza `k` contenente caratteri dall'insieme in input. Ad esempio, `genera(new HashSet<Character>(Arrays.asList('a', 'b', 'c')), 2)` restituisce la lista `[ "aa", "ab", "ac", "ba", "bb", "bc", "ca", "cb", "cc" ]`.

Esercizio 3.3.10. Creare una seconda versione del metodo che prende in input un terzo parametro di tipo `java.util.function.Predicate`. Questa seconda versione del metodo `genera` restituisce tutte le possibili stringhe di lunghezza `k` contenenti caratteri dall'insieme in input per le quali il metodo `test` del predicato passato in input restituisce `true`. Ad esempio:

```
genera(new HashSet<Character>(Arrays.asList('a', 'b', 'c')), 2, s -> s.contains(a));
```

restituisce la lista `[ "aa", "ab", "ac", "ba", "ca" ]`.

Esercizio 3.3.11. Scrivere un metodo che, preso in input un oggetto di tipo `java.io.File`, restituisca una stringa dei file e delle cartelle ivi contenuti. Una cartella è rappresentata dal suo nome seguita da una parentesi aperta, dall'elenco di file e cartelle ivi contenuti separati da spazio e infine da una parentesi chiusa. Ad esempio, la seguente organizzazione:

```
src
   it
      Prova.java
      Test.java
   org
      MyClass.java
   package-info.java
file.txt
```

è rappresentata come segue:

```
"src[ it[ Prova.java Test.java ] org[ MyClass.java ] package-info.java ] file.txt"
```

Metodi utili di `java.io.File` per lo svolgimento dell'esercizio sono: `listFiles, getName, isFile`. Modificare inoltre il metodo in modo che prenda un oggetto di tipo `java.util.function.Predicate` come secondo parametro da utilizzare per filtrare i nomi di `File` da inserire nella rappresentazione risultante.

Esercizio 3.3.12. La distanza di Hamming tra due stringhe di bit entrambe di lunghezza n è il numero di bit in cui le due stringhe differiscono (ad es. `0101` e `0000` hanno distanza di Hamming 2). Implementare un metodo **ricorsivo** che, dato in input un intero k e una stringa di bit s, stampi tutte le stringhe di bit della stessa lunghezza a distanza di Hamming $\leq k$ da s. Ad esempio, se $k = 2$ e $s = 0000$, il metodo stamperà: "`0011 0101 1001 1010 1100`".

Esplicitare chiaramente nel codice il o i casi base e il passo ricorsivo.

Esercizio 3.3.13. Si crei una classe `Punto` costruita con una coppia di interi (x, y). La classe implementa un metodo `contaPercorsi` che, data in input una coppia di interi (a, b) per cui $a \geq x$ e $b \geq y$, restituisca il conteggio di tutti i possibili percorsi da (a, b) a (x, y) spostandosi verso il basso o verso sinistra in direzione di (x, y). Ad esempio, `new Punto(0,0).contaPercorsi(2,1)` restituisce l'intero 3, poiché i percorsi da (2,1) a (0,0) sono: (2,1) → (1,1) → (1,0) → (0,0); (2,1) → (1,1) → (0,1) → (0,0); (2,1) → (2,0) → (1,0) → (0,0).

Esplicitare chiaramente nel codice il/i caso/i base e il passo ricorsivo.

Bonus: Implementare una versione del metodo `contaPercorsi` che stampi o memorizzi l'elenco dei possibili percorsi.

3.4 Stream

Per ognuno dei seguenti esercizi, scrivere il codice pre-Java 8 e in versione Java 8 con lambda, riferimenti a metodi e stream.

Esercizio 3.4.1. Data una lista di stringhe, scrivere un metodo che restituisca una lista delle prime 3 stringhe nella lista di dimensione ≥ 4 e contenenti almeno due caratteri differenti.

Esercizio 3.4.2. Scrivere un metodo che, data in input una lista di stringhe, restituisca l'insieme delle lunghezze delle stringhe nella lista.

Esercizio 3.4.3. Scrivere un metodo che, data in input una lista di stringhe, restituisca l'insieme ordinato della lunghezza delle stringhe nella lista.

Esercizio 3.4.4. Scrivere un metodo che, data in input una lista di stringhe, restituisca una mappa da lunghezza a stringhe di tale lunghezza, con chiavi ordinate per lunghezza.

Esercizio 3.4.5. Scrivere un metodo che, data in input una lista di interi, ne restituisca la somma.

Esercizio 3.4.6. Scrivere un metodo che, data in input una lista di interi, restituisca la concatenazione dei soli quadrati maggiori di 5 sotto forma di stringa, utilizzando la virgola come separatore (ad es., `[2, 10, 3] -> "100,9"`).

Esercizio 3.4.7. Scrivere un metodo che, data in input una lista di stringhe, restituisca una mappa da lunghezza della stringa all'insieme (non la lista) di tutte le stringhe di tale lunghezza.

Esercizio 3.4.8. Scrivere un metodo che, data in input una lista di stringhe (contenenti rappresentazioni di interi sotto forma di stringa), restituisca una mappa da lunghezza della stringa a tutti gli interi corrispondenti la cui stringa ha tale lunghezza.

Esercizio 3.4.9. Siano date la seguente enumerazione e la seguente classe:

```
public enum TipoRistorante
{
   PIZZERIA,
   RISTO,
   BISTRO,
   VEGETARIANO
}

public class Ristorante
{
   private String nome;
   private TipoRistorante tipo;
   private int coperti;

   public Ristorante(String nome, TipoRistorante tipo, int coperti)
   {
      this.nome = nome; this.coperti = coperti; this.tipo = tipo;
   }

   public String getNome() { return nome; }
   public TipoRistorante getTipo() { return tipo; }
```

```
   public int getCoperti() { return coperti; }

   @Override
   public String toString() { return nome+":"+tipo+":"+coperti; }
}
```

Sia data inoltre la seguente lista di ristoranti:

```
List<Ristorante> risto = Arrays.asList(
    new Ristorante("La pergola", TipoRistorante.RISTO, 55),
    new Ristorante("L'etico", TipoRistorante.PIZZERIA, 25),
    new Ristorante("Da Rossi", TipoRistorante.RISTO, 47),
    new Ristorante("Da Gigi", TipoRistorante.PIZZERIA, 42),
    new Ristorante("Giggetto", TipoRistorante.PIZZERIA, 80),
    new Ristorante("Da Ivo", TipoRistorante.PIZZERIA, 150),
    new Ristorante("Romolo e Luigi", TipoRistorante.PIZZERIA, 50),
    new Ristorante("La terrazza", TipoRistorante.RISTO, 40)
    );
```

Scrivere un metodo che stampi una riga per ogni ristorante con nome e numero coperti, in ordine decrescente di numero di coperti.

Esercizio 3.4.10. Data la lista di ristoranti fornita sopra, scrivere un metodo che restituisca l'insieme dei ristoranti che hanno almeno 45 coperti.

Esercizio 3.4.11. Data la lista di ristoranti fornita sopra, scrivere un metodo che restituisca una mappa tipo di ristorante $\rightarrow$ lista dei ristoranti di quel tipo (nella versione Java 8 si utilizzi per la mappatura un riferimento a metodo).

Esercizio 3.4.12. Come nell'esercizio precedente, ma con ciascuna lista di ristoranti ordinata per numero di coperti.

Esercizio 3.4.13. Data la lista di ristoranti fornita sopra, scrivere un metodo che stampi i nomi dei ristoranti in ordine alfabetico separati da virgola.

Esercizio 3.4.14. Data la lista di ristoranti fornita sopra, scrivere un metodo che restituisca la somma totale di tutti i coperti dei ristoranti nella lista.

Esercizio 3.4.15. Scrivere un metodo che stampi i primi 10000 interi (per la versione Java 8 si utilizzi `IntStream.rangeClosed` oppure `Stream.iterate`).

Esercizio 3.4.16. Scrivere un metodo che restituisca l'insieme degli interi da 10 a 100.

Esercizio 3.4.17. Scrivere un metodo che restituisca la lista degli interi da 100 a 10, con passo due (100, 98, 96, ecc. fino a 10).

Esercizio 3.4.18. Scrivere un metodo che legga tutte le parole di un file (separate da spazio) e restituisca una mappa parola → conteggio.

Esercizio 3.4.19. Siano date la seguente classe:

```
public class Titolo
{
   public enum Allineamento { CX, SX, DX }
   private Allineamento allineamento;
   private List<Riga> righe;

   public Titolo(Allineamento a) { this(a, new ArrayList<>()); }
   public Titolo(Allineamento a, List<Riga> righe)
   {
      allineamento = a; this.righe = righe;
   }
   public void add(Riga r) { righe.add(r); }
   public boolean isCentered() { return allineamento == Allineamento.CX; }
   @Override public String toString() { return righe.toString(); }
   public Allineamento getAllineamento() { return allineamento; }
   public List<Riga> getRighe() { return new ArrayList<>(righe); }

   static public class Riga
   {
      private String riga;
      private int numero;

      public Riga(String riga, int numero) { this.riga = riga; this.numero = numero; }
      public Riga(String riga) { this(riga, -1); }
      @Override public String toString() { return (numero == -1 ? "" : numero+": ")+riga; }
   }
}
```

A partire da una `List<Titolo>` lista, si calcolino senza precalcoli di strutture dati intermedie (ovvero, su una sola riga, utilizzando gli stream):

- l'insieme dei primi 5 titoli con al più una riga
- la lista dei soli titoli centrati e in ordine alfabetico
- la mappa da allineamento a lista di titoli
- la mappa da allineamento a insieme di titoli
- la mappa da allineamento alla concatenazione delle stringhe dei titoli
- la mappa da allineamento alla lista delle stringhe dei titoli
- l'insieme delle righe dei titoli nella lista sotto forma di stringhe
- la mappa dei conteggi delle parole nelle righe di titoli calcolate al punto precedente (questo esercizio richiede l'utilizzo di `flatMap`)
  1. Utilizzando `Collectors.groupingBy`
  2. Utilizzando `Collectors.toMap`

3.5 Design Pattern

Esercizio 3.5.1. Si consideri il seguente codice:

```
public class Configuration
{
    public final static String FILENAME = "config/project.properties";
    private Map<String, String> config = new HashMap<>();

    public Configuration()
    {
        BufferedReader br = null;

        try
        {
            br = new BufferedReader(new FileReader(FILENAME));
            while(br.ready())
            {
                String[] pair = br.readLine().split("=");
                config.put(pair[0], pair[1]);
            }
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }

        // ...
    }

    public String get(String name) { return config.get(name); }
}
```

Modificare la classe utilizzando un design pattern appropriato in modo tale che la stessa, unica configurazione sia accessibile da qualsiasi classe del progetto.

Esercizio 3.5.2. Uno sviluppatore inesperto ha creato la seguente classe `Lessico`:

```
public class Lessico
{
   final static public Path DEFAULT_LEXICON = Paths.get("lexicon.txt");
   private Set<String> lessico = new HashSet<>();

   public Lessico() throws IOException
   {
      this(DEFAULT_LEXICON);
   }

   public Lessico(Path p) throws IOException
   {
      Files.lines(p).forEach(lessico::add);
   }
```

```
   public boolean isWord(String w) { return lessico.contains(w); }
}
```

Il problema della classe è che, per ogni istanza di `Lessico`, si carica in memoria un nuovo insieme di termini anche se il `Path` è il medesimo, riempiendo inutilmente la memoria RAM disponibile.

Si modifichi la classe **correggendola** mediante un design pattern che eviti di creare istanze multiple di lessici ottenuti dallo stesso file. E' possibile sviluppare una versione semplificata in cui si evita di creare duplicati per il solo lessico di default.

Esercizio 3.5.3. Un bottone grafico vuole comunicare l'evento di pressione del bottone a tutti coloro che vogliono ricevere tale informazione. Si utilizzi un design pattern adeguato (implementandolo per conto proprio o utilizzando/estendendo le classi della libreria standard di Java) per sviluppare il meccanismo di comunicazione tra bottone e classi interessate. A mo' di esempio, si implementi quale classe interessata una classe astratta `Esecutore` che, ricevendo l'informazione che il bottone è stato premuto, richiami il metodo `esegui` che essa espone (ma non implementa).

Esercizio 3.5.4. Si consideri la seguente classe:

```
public class Sequenza
{
   public enum AlgoritmoOrdinamento { BUBBLESORT, QUICKSORT, }

   public void ordina(AlgoritmoOrdinamento o)
   {
      switch(o)
      {
         case BUBBLESORT: ordinaConBubbleSort(); break;
         case QUICKSORT: ordinaConQuickSort(); break;
      }
   }

   private void ordinaConBubbleSort() { /* ... */ }
   private void ordinaConQuickSort() { /* ... */ }
}
```

Utilizzare lo Strategy Pattern per modificare la classe in modo da:

- eliminare l'enumeration e rendere variabile il tipo di algoritmo di ordinamento da utilizzare
- permettere l'impostazione dell'algoritmo di ordinamento da utilizzare mediante un metodo `setAlgoritmoDiOrdinamento`

Non è richiesta l'implementazione degli algoritmi di ordinamento.

Esercizio 3.5.5. Si consideri il seguente codice:

```
abstract class Bottone
{
    abstract public void esegui();
}

class Bottone1 extends Bottone
{
    public void esegui() { System.out.println("ciao"); }
}

class Bottone2 extends Bottone
{
    private int k, j;
    private String label;
    public Bottone2 (int k, int j) { this.k = k; this.j = j; }
    public void esegui() { label = ""+k/j; }
}
```

Modificare la classe utilizzando il Command Pattern in modo tale da incapsulare l'azione associata ad un `Bottone` tramite il metodo `esegui` e renderla parametrizzata.

Esercizio 3.5.6. Si consideri il seguente codice:

```
public class String2Integer
{
    private String s;

    public String2Integer(String s) { this.s = s; }

    public int toParsedInteger() { return Integer.parseInt(s); }
    public int toIntegerSize() { return s.length(); }
    public int toIntegerCharSum() { return s.chars().reduce(0, (x, y) -> x+y ); }
}
```

Utilizzare lo Strategy Pattern per modificare la classe in modo da:

- sostituire i tre metodi di trasformazione stringa-intero con un unico metodo `toInteger` il cui comportamento sia parametrizzato
- permettere l'impostazione dinamica della trasformazione stringa-intero da utilizzare mediante un metodo `setIntegerFunction`

Esercizio 3.5.7. Utilizzare il Builder pattern per gestire dinamicamente la costruzione di una lista di interi. Si richiede che il builder disponga dei seguenti comportamenti:

- Un metodo `add(Integer... l)` che aggiunge una sequenza di interi alla lista
- Un metodo `removeAll(Predicate<Integer> p)` che rimuove tutti gli elementi per i quali il predicato `p` è vero

- Un metodo `times(int k)` che, dato un intero `k`, replica la lista `k` volte
- Un metodo `build()` che genera la lista di interi così configurata e la restituisce

Esercizio 3.5.8. Utilizzare il Builder pattern per gestire la costruzione *lazy* di una stringa cui vengono applicare una serie di funzioni di trasformazione. Si richiede che il builder disponga dei seguenti comportamenti:

- Un metodo `perform(Function<String, String>)` che dispone l'applicazione futura della funzione alla stringa, per trasformarla di conseguenza;
- Un metodo `build()` che genera la stringa risultante dall'applicazione delle funzioni specificate.

Ad esempio, il seguente codice:

```
String s = new StringBuilder("ciao")
               .perform(x -> x.substring(1))
               .perform(x -> "m"+x)
               .perform(x -> "--"+x+"--").build();
System.out.println(s);
```

stamperà `--miao--`.

Esercizio 3.5.9. Utilizzare il Simple Factory pattern per gestire la costruzione di giocattoli. La superclasse astratta `Giocattolo` viene estesa da `Console` e `Cicciobello`. La factory espone un metodo `getGiocattolo` che permette la costruzione del giocattolo richiesto sulla base del nome del giocattolo. Si lanci un'eccezione personalizzata (con il nome del giocattolo richiesto) nel caso in cui non sia possibile procedere alla costruzione dell'oggetto.