

Esercizi di Java Avanzato

Marco Faella

9 luglio 2015

Capitolo 1

Esercizi

1. (20 punti, 11 febbraio 2013) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
abstract class A {
    public abstract String f(A a, B b);
    public int f(B b, C c) { return 1; }
}
class B extends A {
    public String f(A a, B b) { return "2"; }
    public String f(C c, B b) { return "3"; }
    public int f(C c, Object x) { return 4; }
}
class C extends B {
    public String f(C c1, C c2) { return "5"; }
    public String f(A a, B b) { return "6"; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(null, gamma));
        System.out.println(beta.f(gamma, gamma));
        System.out.println(beta.f(gamma, alfa));
        System.out.println(gamma.f(beta, beta));
        System.out.println(1 + "1");
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
2. **MultiSet.** (35 punti, 11 febbraio 2013) Un `MultiSet` è un insieme in cui ogni elemento può comparire più volte. Quindi, ammette duplicati come una lista, ma, a differenza di una lista, l'ordine in cui gli elementi vengono inseriti non è rilevante. Implementare una classe parametrica `MultiSet`, con i seguenti metodi:
- `add`, che aggiunge un elemento,
 - `remove`, che rimuove un elemento (se presente), ed
 - `equals`, che sovrascrive quello di `Object` e considera uguali due `MultiSet` se contengono gli stessi elementi, ripetuti lo stesso numero di volte.

Infine, deve essere possibile iterare su tutti gli elementi di un `MultiSet` usando il ciclo `for-each`.

Esempio d'uso:	Output (l'ordine dei numeri è irrilevante):
<pre>MultiSet<Integer> s1 = new MultiSet<Integer>(); MultiSet<Integer> s2 = new MultiSet<Integer>(); s1.add(5); s1.add(7); s1.add(5); s2.add(5); s2.add(5); s2.add(7); for (Integer n: s1) System.out.println(n); System.out.println(s1.equals(s2));</pre>	<pre>7 5 5 true</pre>

3. **Concurrent filter.** (35 punti, 11 febbraio 2013) Data la seguente interfaccia:

```
public interface Selector<T> {
    boolean select(T x);
}
```

implementare il metodo (statico) `concurrentFilter`, che prende come argomenti un `Set X` e un `Selector S`, di tipi compatibili, e restituisce un nuovo insieme `Y` che contiene quegli elementi di `X` per i quali la funzione `select` di `S` restituisce il valore `true`.

Inoltre, il metodo deve invocare la funzione `select` in parallelo su tutti gli elementi di `X` (dovrà quindi creare tanti thread quanti sono gli elementi di `X`).

Esempio d'uso:	Output:
<pre>Set<Integer> x = new HashSet<Integer>(); x.add(1); x.add(2); x.add(5); Selector<Integer> oddSelector = new Selector<Integer>() { public boolean select(Integer n) { return (n%2 != 0); } }; Set<Integer> y = concurrentFilter(x, oddSelector); for (Integer n: y) System.out.println(n);</pre>	<pre>1 5</pre>

4. (10 punti, 11 febbraio 2013) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Il pattern `Composite` prevede che gli oggetti primitivi abbiano un riferimento al contenitore in cui sono inseriti.
- ☐ ☐ Il pattern `Composite` prevede che un contenitore si possa comportare come un oggetto primitivo.
- ☐ ☐ Nel framework `MVC`, ogni oggetto *view* comunica con almeno un oggetto *model*.
- ☐ ☐ La scelta del layout di un container `AWT` rappresenta un'istanza del pattern `Strategy`.
- ☐ ☐ Il pattern `Observer` prevede un'interfaccia che sarà implementata da tutti gli osservatori.

5. (25 punti, 3 settembre 2012) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, B y) { return "A1"; }
    public String f(B x, C y) { return "A2"; }
}
class B extends A {
    public String f(B x, C y) { return "B1:" + f(x, x); }
    public String f(B x, B y) { return "B2"; }
```

```

}
class C extends B {
    public String f(A x, A y) { return "C1:" + y.f(y, null); }
    public String f(B x, B y) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = beta;
        System.out.println(beta.f(gamma, beta));
        System.out.println(gamma.f(beta, gamma));
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
6. **Mystery thread 2.** (25 punti, 3 settembre 2012) Elencare tutte le sequenze di output possibili per il seguente programma.

```

public static void main(String[] args) {
    class MyThread extends Thread {
        private int id;
        private Thread other;
        public MyThread(int n, Thread t) {
            id = n;
            other = t;
        }
        public void run() {
            try {
                if (other != null)
                    other.join();
            } catch (InterruptedException e) {
                return;
            }
            System.out.println(id);
        }
    }
    Thread t1 = new MyThread(1, null);
    Thread t2 = new MyThread(2, null);
    Thread t3 = new MyThread(3, t1);
    Thread t4 = new MyThread(4, t2);
    t1.start(); t2.start(); t3.start(); t4.start();
}

```

7. **Anagrammi.** (32 punti, 3 settembre 2012) Implementare la classe `MyString`, che rappresenta una stringa con la seguente caratteristica: due oggetti `MyString` sono considerati uguali (da `equals`) se sono uno l'anagramma dell'altro. Inoltre, la classe `MyString` deve essere clonabile e deve offrire un'implementazione di `hashCode` coerente con `equals` e non banale (che non restituisca lo stesso codice hash per tutti gli oggetti).

Suggerimento: Nella classe `String` è presente il metodo `public char charAt(int i)`, che restituisce l'i-esimo carattere della stringa, per i compreso tra 0 e `length()-1`.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> MyString a = new MyString("uno_due_tre"); MyString b = new MyString("uno_tre_deu"); MyString c = new MyString("ert_unodue"); MyString d = c.clone(); System.out.println(a.equals(b)); System.out.println(b.equals(c)); System.out.println(a.hashCode()==b.hashCode()); </pre>	<pre> true false true </pre>

8. **Bijection.** (23 punti, 3 settembre 2012) Implementare la classe parametrica `Bijection`, che rappresenta una biezione (relazione biunivoca) tra un insieme di chiavi e uno di valori. Il metodo `addPair` aggiunge una coppia chiave-valore alla relazione. Se la chiave oppure il valore sono già presenti, viene lanciata un'eccezione. Il metodo `getValue` accetta come argomento una chiave e restituisce il valore associato, oppure `null` se la chiave non è presente. Il metodo `getKey` accetta un valore e restituisce la chiave corrispondente, oppure `null` se il valore non è presente. Entrambi i metodi `getValue` e `getKey` devono operare in tempo costante.

Esempio d'uso:	Output:
<pre> Bijection<Integer,String> b = new Bijection<Integer,String>(); b.addPair(12,"dodici"); b.addPair(7,"sette"); System.out.println(b.getValue(12)); System.out.println(b.getKey("tredici")); b.addPair(8,"sette"); </pre>	<pre> dodici null Exception in thread "main" ... </pre>

9. (20 punti, 3 settembre 2012) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ L'operatore `.class` si applica ad un riferimento
- ☐ ☐ Tramite riflessione è possibile invocare un metodo il cui nome è sconosciuto a tempo di compilazione
- ☐ ☐ `ArrayList<Integer>` è sottotipo di `Set<?>`
- ☐ ☐ `ArrayList<Integer>` è sottotipo di `List<? extends Number>`
- ☐ ☐ L'eccezione `ArrayIndexOutOfBoundsException` è verificata (*checked*)
- ☐ ☐ Nel pattern `Observer`, il soggetto osservato ha dei riferimenti ai suoi osservatori
- ☐ ☐ La ridefinizione (*overriding*) del metodo `equals` da parte di una classe rappresenta un esempio del pattern `Strategy`
- ☐ ☐ Nel pattern `Decorator`, l'oggetto decorato ha dei riferimenti agli oggetti decoratori
- ☐ ☐ Il metodo `toString` della classe `Object` rappresenta un esempio del pattern `Factory Method`
- ☐ ☐ Il pattern `Strategy` e il pattern `Template Method` sono soluzioni alternative allo stesso problema.

10. (20 punti, 22 gennaio 2013) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(A x, A y) { return "A1"; }
    private String f(A x, Object y) { return "A2"; }
}
class B extends A {
    public String f(B x, A y) { return "B1:" + y.f(y, y); }
    public String f(A x, A y) { return "B2"; }
}

```

```

}
class C extends B {
    public String f(A x, A y) { return "C1:" + y.f(y, null); }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = new A();
        System.out.println(beta.f(gamma, alfa));
        System.out.println(gamma.f(alfa, alfa));
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
11. **Shared object.** (20 punti, 22 gennaio 2013) Elencare tutte le sequenze di output possibili per il seguente programma.

```

public static void main(String[] args) throws InterruptedException {
    class MyThread extends Thread {
        private int id;
        private int[] arr;
        public MyThread(int id, int[] arr) {
            this.id = id;
            this.arr = arr;
        }
        public void run() {
            synchronized (arr) {
                arr[0]++;
                System.out.println(id + ":" + arr[0]);
            }
            return;
        }
    }
    int[] a = { 0 };
    Thread t1 = new MyThread(1,a);
    Thread t2 = new MyThread(2,a);
    Thread t3 = new MyThread(3,a);
    t1.start(); t2.start(); t3.start();
}

```

12. **Insieme di lettere.** (28 punti, 22 gennaio 2013) La classe `MyString` rappresenta una stringa. Due oggetti di tipo `MyString` sono considerati uguali (da `equals`) se utilizzano le stesse lettere, anche se in numero diverso. Ad esempio, “casa” è uguale a “cassa” e diverso da “sa”; “culle” è uguale a “luce” e diverso da “alluce”. La classe `MyString` deve essere clonabile e deve offrire un'implementazione di `hashCode` coerente con `equals` e non banale (che non restituisca lo stesso codice hash per tutti gli oggetti).

Suggerimento: Nella classe `String` è presente il metodo `public char charAt(int i)`, che restituisce l'i-esimo carattere della stringa, per i compreso tra 0 e `length()-1`.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> MyString a = new MyString("freddo"); MyString b = new MyString("defro"); MyString c = new MyString("caldo"); MyString d = c.clone(); System.out.println(a.equals(b)); System.out.println(b.equals(c)); System.out.println(a.hashCode()==b.hashCode()); </pre>	<pre> true false true </pre>

13. **MaxBox.** (22 punti, 22 gennaio 2013) Implementare la classe parametrica **MaxBox**, che funziona come un contenitore che conserva solo l'elemento "massimo" che vi sia mai stato inserito. L'ordinamento tra gli elementi può essere stabilito in due modi diversi: se al costruttore di **MaxBox** viene passato un oggetto **Comparator**, quell'oggetto verrà usato per stabilire l'ordinamento; altrimenti, verrà utilizzato l'ordinamento naturale fornito da **Comparable**. In quest'ultimo caso, se la classe degli elementi non implementa **Comparable**, viene sollevata un'eccezione.

Esempio d'uso:	Output:
<pre> MaxBox<String> bb1 = new MaxBox<String>(); MaxBox<String> bb2 = new MaxBox<String>(new Comparator<String>() { public int compare(String a, String b) { return a.length() - b.length(); } }); bb1.insert("dodici"); bb1.insert("sette"); bb2.insert("dodici"); bb2.insert("sette"); System.out.println(bb1.getMax()); System.out.println(bb2.getMax()); </pre>	<pre> sette dodici </pre>

14. (10 punti, 22 gennaio 2013) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Nel pattern **Decorator**, l'oggetto decoratore si comporta come l'oggetto da decorare.
- ☐ ☐ Nel pattern **Decorator**, l'oggetto da decorare ha un metodo che aggiunge una decorazione.
- ☐ ☐ Il metodo **Collections.sort** rappresenta un'istanza del pattern **Template Method**.
- ☐ ☐ Secondo il pattern **Observer**, gli osservatori devono contenere un riferimento all'oggetto osservato.
- ☐ ☐ Il pattern **Strategy** permette di fornire versioni diverse di un algoritmo.

15. (25 punti, 18 giugno 2012) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(A x, int n) { return "A1:" + n; }
    public String f(A x, double n) { return "A2:" + n; }
}
class B extends A {
    public String f(A x, int n) { return "B1:" + n; }
    public String f(B x, Object o) { return "B2"; }
}
class C extends B {
    public String f(A x, int n) { return "C1:" + n; }
}

```



```

    public String f(C x, double n) { return "C2:" + f(x, x); }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = new B();
        System.out.println(beta.f(null, 7));
        System.out.println(alfa.f(gamma, 5));
        System.out.println(gamma.f(beta, 3.0));
        System.out.println((1 < 1) > 1);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

16. **Point.** (25 punti, 18 giugno 2012) La classe `Point` rappresenta un punto del piano cartesiano con coordinate intere:

```

public class Point {
    private int x, y;
    ...
}

```

Spiegare quali delle seguenti sono implementazioni valide per il metodo `compare(Point a, Point b)` tratto dall'interfaccia `Comparator<Point>`, supponendo che tale metodo abbia accesso ai campi privati di `Point`.

- (a) `return a.x-b.x;`
- (b) `return a.x-b.y;`
- (c) `return ((a.x*a.x)+(a.y*a.y)) - ((b.x*b.x)+(b.y*b.y));`
- (d) `return (a.x-b.x)+(a.y-b.y);`
- (e) `return (a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y);`

17. **BoundedMap.** (35 punti, 18 giugno 2012) Implementare la classe `BoundedMap`, che rappresenta una mappa con capacità limitata. Il costruttore accetta la dimensione massima della mappa. I metodi `get` e `put` sono analoghi a quelli dell'interfaccia `Map`. Se però la mappa è piena e viene invocato il metodo `put` con una chiave nuova, verrà rimossa dalla mappa la chiave che fino a quel momento è stata ricercata meno volte con `get`. L'implementazione deve rispettare il seguente caso d'uso.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> BoundedMap<String,String> m = new BoundedMap<String,String> >(2); m.put("NA", "Napoli"); m.put("SA", "Salerno"); System.out.println(m.get("NA")); m.put("AV", "Avellino"); System.out.println(m.get("SA")); </pre>	<pre> Napoli null </pre>

18. **ThreadRace.** (25 punti, 18 giugno 2012) Implementare il metodo statico `threadRace`, che accetta due oggetti `Runnable` come argomenti, li esegue contemporaneamente e restituisce 1 oppure 2, a seconda di quale dei due `Runnable` è terminato prima.

Si noti che `threadRace` è un metodo bloccante. Sarà valutato negativamente l'uso di attesa attiva.

19. (20 punti, 18 giugno 2012) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Le enumerazioni estendono implicitamente la classe `Enum`.
- ☐ ☐ `Comparable<Integer>` è sottotipo di `Comparable<Number>`.
- ☐ ☐ `Integer` è sottotipo di `Number`.
- ☐ ☐ `Cloneable` è una interfaccia vuota.
- ☐ ☐ Due oggetti con lo stesso *hash code* dovrebbero essere considerati uguali da *equals*.
- ☐ ☐ Il pattern **Strategy** suggerisce di utilizzare un oggetto per rappresentare un algoritmo.
- ☐ ☐ Il modo in cui si associa un gestore di eventi alla pressione di un `JButton` in `Swing`/`AWT` rappresenta un'istanza del pattern **Strategy**.
- ☐ ☐ Il pattern `Iterator` prevede un metodo per far ripartire l'iteratore daccapo.
- ☐ ☐ Il pattern `Composite` prevede un metodo per distinguere un oggetto primitivo da uno composito.
- ☐ ☐ Il pattern `Decorator` prevede che si possa utilizzare un oggetto decorato nello stesso modo di uno non decorato.

20. (31 punti, 22 marzo 2013) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
abstract class A {
    public abstract String f(A a, B b);
    public int f(B b, C c) { return 1; }
}
class B extends A {
    public String f(A a, B b) { return "2"; }
    public int f(B c, C b) { return 3; }
    public int f(C c, Object x) { return 4; }
}
class C extends B {
    public String f(C c1, C c2) { return "5"; }
    public String f(A a, B b) { return "6"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(beta, null));
        System.out.println(beta.f(beta, beta));
        System.out.println(beta.f(gamma, alfa));
        System.out.println(gamma.f(gamma, gamma));
        System.out.println(beta.getClass().getName());
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

21. **Auditorium.** (30 punti, 22 marzo 2013) La seguente classe (semplificata) `Seat` rappresenta un posto in un auditorium.

```
public class Seat { public int row, col; }
```

La classe `Auditorium` serve ad assegnare i posti in un teatro. Il costruttore prende come argomenti le dimensioni della platea, in termini di file e posti per fila, nonché un oggetto `Comparator` che serve ad ordinare i posti in ordine di preferenza. Il metodo `assignSeats` prende come argomenti il numero n di posti richiesti e restituisce un insieme contenente gli n posti migliori (in base al comparatore) ancora disponibili. Se la platea non contiene n posti disponibili, il metodo restituisce `null`.

Esempio d'uso:	Output:
<pre>Auditorium a = new Auditorium(5, 5, new Comparator<Seat>() { public int compare(Seat a, Seat b) { return (a.row==b.row)? (a.col-b.col): (a.row-b.row); } }); Set<Seat> s = a.assignSeats(4); System.out.println(s);</pre>	<pre>[(0,0), (0,1), (0,2), (0,3)]</pre>

22. **Cane.** (21 punti, 22 marzo 2013) Data la seguente classe:

```
public class Cane {
    private Person padrone;
    private String nome;
    ...
}
```

Si considerino le seguenti specifiche alternative per il metodo `equals`. Due oggetti `x` e `y` di tipo `Cane` sono uguali se:

- (a) `x` e `y` non hanno padrone (cioè, `padrone == null`) e hanno lo stesso nome;
 - (b) `x` e `y` hanno lo stesso padrone oppure lo stesso nome;
 - (c) `x` e `y` hanno due nomi di pari lunghezza (vale anche `null` o stringa vuota);
 - (d) `x` e `y` non hanno padrone **oppure** entrambi hanno un padrone ed i loro nomi iniziano con la stessa lettera (vale anche `null` o stringa vuota).
- Dire quali specifiche sono valide, fornendo un controesempio in caso negativo. (16 punti)
 - Implementare la specifica (c). (5 punti)

23. **Shared average.** (33 punti, 22 marzo 2013) La classe `SharedAverage` permette a diversi thread di comunicare un valore numerico (`double`) e poi ricevere il valore medio tra tutti quelli inviati dai diversi thread. Precisamente, il costruttore accetta come argomento il numero n di thread che parteciperà all'operazione. Il metodo `sendValAndReceiveAvg` accetta come argomento il valore indicato dal thread corrente, mette il thread corrente in attesa che tutti gli n thread abbiano inviato un valore, e infine restituisce la media aritmetica di tutti i valori inviati.

Se un thread ha già chiamato `sendValAndReceiveAvg` una volta, al secondo tentativo viene sollevata un'eccezione.

È necessario evitare *race condition*.

Esempio d'uso:	Comportamento:
<pre>SharedAverage a = new SharedAverage(10); double average; try { average = a.sendValAndReceiveAvg(5.0); } catch (InterruptedException e) { return; }</pre>	<p>Quando altri 9 thread avranno invocato <code>sendValAndReceiveAvg</code>, tutte le invocazioni restituiranno la media dei 10 valori inviati.</p>

24. (10 punti, 22 marzo 2013) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Il pattern **Observer** permette a diversi oggetti di ricevere gli eventi generati dallo stesso oggetto.
- ☐ ☐ Il pattern **Observer** prevede che osservatore e soggetto osservato implementino una stessa interfaccia.
- ☐ ☐ Il pattern **Composite** consente la composizione ricorsiva di oggetti.
- ☐ ☐ Il pattern **Iterator** permette a diversi thread di iterare contemporaneamente sulla stessa collezione.
- ☐ ☐ I design pattern offrono una casistica dei più comuni errori di progettazione.

25. (24 punti, 23 aprile 2012) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, A[] arr) { return "A1"; }
    public String f(Object x, Object[] y) { return "A2"; }
}
class B extends A {
    public String f(B x, Object[] y) { return "B1->" + f(y, y); }
}
class C extends B {
    public String f(A x, A[] arr) { return "C1"; }
    public String f(Object x, Object y) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        A[] arr = new B[10];
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(beta.f(null, arr));
        System.out.println(gamma.f(arr, alfa));
        System.out.println(gamma.f(alfa, arr));
        System.out.println(1 << 1);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (12 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)

26. **Safe.** (28 punti, 23 aprile 2012) Implementare la classe **Safe**, che rappresenta una cassaforte che contiene una stringa segreta, protetta da un numero intero che funge da combinazione. Il costruttore accetta la combinazione e la stringa segreta. Il metodo **open** accetta un numero intero e restituisce la stringa segreta se tale numero coincide con la combinazione. Altrimenti, restituisce null. Infine, se le ultime 3 chiamate a **open** sono fallite, la cassaforte diventa irreversibilmente **bloccata** ed ogni ulteriore operazione solleva un'eccezione.

Implementare la classe **ResettableSafe** come una sottoclasse di **Safe** che aggiunge il metodo **changeKey**, che accetta due interi *old* e *new* e restituisce un boolean. Se la cassaforte è bloccata, il metodo solleva un'eccezione. Altrimenti, se l'argomento *old* coincide con la combinazione attuale, il metodo imposta la combinazione della cassaforte a *new* e restituisce true. Se invece *old* differisce dalla combinazione attuale, il metodo restituisce false.

Una **ResettableSafe** diventa bloccata dopo tre tentativi falliti di **open** o di **changeKey**. Ogni chiamata corretta a **open** o **changeKey** azzerà il conteggio dei tentativi falliti.

Suggerimento: prestare attenzione alla scelta della visibilità di campi e metodi.

Esempio d'uso:	Output dell'esempio d'uso:
<pre>ResetableSafe s = new ResetableSafe(2381313, "L'assassino_e' _il_maggiordomo."); System.out.println(s.open(887313)); System.out.println(s.open(13012)); System.out.println(s.changeKey(12,34)); System.out.println(s.open(2381313));</pre>	<pre>null null false Exception in thread "main"...</pre>

27. (20 punti, 23 aprile 2012) Con riferimento alla classe **Safe** dell'esercizio 2, dire quali delle seguenti specifiche per il metodo **equals** sono valide e perché.
Due istanze di **Safe** sono uguali se...

- (a) ...hanno il messaggio segreto di pari lunghezza.
- (b) ...hanno la stessa combinazione oppure lo stesso messaggio segreto.
- (c) ...hanno la combinazione maggiore di zero.
- (d) ...la somma delle due combinazioni è un numero pari.
- (e) ...almeno uno dei due messaggi segreti contiene la parola "maggiordomo".

28. **Panino.** (28 punti, 23 aprile 2012) Implementare la classe **Panino**, il cui metodo **addIngrediente** aggiunge un ingrediente, scelto da un elenco fisso. Gli ingredienti sono divisi in categorie. Se si tenta di aggiungere più di un ingrediente della stessa categoria, il metodo **addIngrediente** solleva un'eccezione.

Elenco delle categorie e degli ingredienti:

ripieni: PROSCIUTTO, SALAME

formaggi: SOTTILETTA, MOZZARELLA

salse: MAIONESE, SENAPE

Esempio d'uso:	Output dell'esempio d'uso:
<pre>Panino p = new Panino(); p.addIngrediente(Panino.Ingrediente.SALAME); p.addIngrediente(Panino.Ingrediente.SOTTILETTA); System.out.println(p); p.addIngrediente(Panino.Ingrediente.MOZZARELLA);</pre>	<pre>panino con SALAME, SOTTILETTA Exception in thread "main"...</pre>

29. **Mystery thread.** (25 punti, 9 luglio 2012) Escludendo i cosiddetti *spurious wakeup* (risvegli da wait senza che sia avvenuta una notify), elencare tutte le sequenze di output possibili per il seguente programma.

```
public static void main(String[] args) throws InterruptedException {
    final Object dummy = new Object();
    final Thread t1 = new Thread() {
        public void run() {
            synchronized (dummy) {
                while (true) {
                    try { dummy.wait(); System.out.println("T1:A"); }
                    catch (InterruptedException e) { break; }
                }
                System.out.println("T1:B");
            }
        }
    };
    final Thread t2 = new Thread() {
```

```

        public void run() {
            synchronized (dummy) {
                dummy.notifyAll();
                System.out.println("T2:A");
            }
            t1.interrupt();
            System.out.println("T2:B");
        }
    };
    t1.start();
    t2.start();
    t1.join();
    System.out.println("Fine");
}

```

30. (25 punti, 9 luglio 2012) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(A x, B y) { return "A1"; }
    public String f(B x, C y) { return "A2"; }
}
class B extends A {
    public String f(B x, C y) { return "B1:" + f(x, x); }
    public String f(B x, B y) { return "B2"; }
}
class C extends B {
    public String f(A x, B y) { return "C1"; }
    public String f(B x, B y) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = beta;
        System.out.println(beta.f(null, alfa));
        System.out.println(gamma.f(beta, gamma));
        System.out.println(alfa.f(gamma, beta));
        System.out.println((1 >> 1) < 0);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
31. **Social network.** (33 punti, 9 luglio 2012) Nell'ambito dell'implementazione di un *social network*, la classe `Person` rappresenta un utente. Tramite i metodi `addFriend` e `addEnemy` è possibile aggiungere un amico o un nemico a questa persona, rispettando le seguenti regole:

- (a) una persona non può aggiungere se stessa come amico o nemico;
- (b) una persona non può aggiungere la stessa persona sia come amico sia come nemico.

Il metodo `contacts` permette di iterare su tutti i contatti di questa persona tramite un iteratore, che restituirà prima tutti gli amici e poi tutti i nemici.

Esempio d'uso: <pre> Person a = new Person("Antonio"); Person c = new Person("Cleopatra"); Person o = new Person("Ottaviano"); a.addEnemy(o); a.addFriend(c); for (Person p: a.contacts()) System.out.println(p); </pre>	Output dell'esempio d'uso: Cleopatra Ottaviano
---	--

32. **NumberType.** (22 punti, 9 luglio 2012) Implementare la classe enumerata `NumberType`, che rappresenta i sei tipi primitivi numerici del linguaggio Java. Il campo `width` contiene l'ampiezza di questo tipo, in bit. Il metodo `isAssignableTo` prende come argomento un'altra istanza `t` di `NumberType` e restituisce vero se questo tipo è assegnabile a `t` (ovvero, c'è una conversione implicita dal tipo rappresentato da `this` al tipo rappresentato da `t`) e falso altrimenti.

Esempio d'uso: <pre> System.out.println(NumberType.SHORT.width); System.out.println(NumberType.INT.isAssignableTo(NumberType.FLOAT)); </pre>	Output: 16 true
--	---------------------------

33. (20 punti, 9 luglio 2012) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Una classe astratta può avere metodi concreti (dotati di corpo).
- ☐ ☐ Una classe interna può essere `protected`.
- ☐ ☐ `ArrayIndexOutOfBoundsException` è una eccezione verificata (*checked*).
- ☐ ☐ `RandomAccess` è una interfaccia parametrica.
- ☐ ☐ La classe `Thread` ha un costruttore senza argomenti.
- ☐ ☐ Nel pattern `Observer`, più oggetti possono osservare lo stesso oggetto.
- ☐ ☐ Il pattern `Observer` prevede un'interfaccia che sarà implementata da tutti gli osservatori.
- ☐ ☐ Il pattern `Template Method` suggerisce l'utilizzo di una classe astratta.
- ☐ ☐ Il pattern `Composite` organizza degli oggetti in una gerarchia ad albero.
- ☐ ☐ Nel framework `MVC`, le classi `Controller` si occupano dell'interazione con l'utente.

34. (25 punti, 24 giugno 2015) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(A x, A y, B z) { return "A1"; }
    public String f(A x, Object y, B z) { return "A2"; }
    public String f(B x, Object y, B z) { return "A3"; }
}
class B extends A {
    public String f(A x, A y, B z) { return "B1"; }
    public String f(A x, Object y, A z) { return "B2"; }
    public String f(A x, Object y, B z) { return "B3"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;

        System.out.println(alfa.f(alfa, alfa, null));
    }
}

```

```

        System.out.println(beta.f(alfa, beta, beta));
        System.out.println(beta.f(beta, beta, beta));
        System.out.println(beta.f(alfa, alfa, alfa));
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

35. **SimpleThread.** (20 punti, 24 giugno 2015) Indicare tutti gli output possibili di un programma che faccia partire contemporaneamente due istanze della seguente classe `SimpleThread`.

```

public class SimpleThread extends Thread
{
    private static volatile int n = 0;

    public void run() {
        n++;
        int m = n;
        System.out.println(m);
    }
}

```

36. **Controller.** (45 punti, 24 giugno 2015) Realizzare la classe `Controller`, che rappresenta una centralina per autoveicoli, e la classe `Function`, che rappresenta una funzionalità del veicolo, che può essere accesa o spenta. Alcune funzionalità sono *incompatibili* tra loro, per cui accenderne una fa spegnere automaticamente l'altra.

La classe `Controller` ha due metodi: `addFunction` aggiunge al sistema una nuova funzionalità con un dato nome; `printOn` stampa a video i nomi delle funzionalità attive. La classe `Function` ha tre metodi: `turnOn` e `turnOff` per attivarla e disattivarla; `setIncompatible` accetta un'altra funzionalità x e imposta un'incompatibilità tra *this* e x .

Leggere attentamente il seguente caso d'uso, che mostra, tra le altre cose, che l'incompatibilità è automaticamente simmetrica, ma *non* transitiva.

Caso d'uso:

```

Controller c = new Controller();
Controller.Function ac = c.addFunction("Aria_condizionata");
Controller.Function risc = c.addFunction("Riscaldamento");
Controller.Function sedile = c.addFunction("Sedile_riscaldato");

ac.setIncompatible(risc);
ac.setIncompatible(sedile);

ac.turnOn();
c.printOn();
System.out.println("----");

risc.turnOn();
sedile.turnOn();
c.printOn();

```

Output:

```

Aria condizionata
----
Sedile riscaldato
Riscaldamento

```

37. **listIntersection.** (25 punti, 24 giugno 2015) Implementare il metodo statico `listIntersection`, che accetta una lista e un insieme, e restituisce una nuova lista, che contiene gli elementi della lista che appartengono anche all'insieme, nello stesso ordine in cui appaiono nella prima lista.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporre un'altra.

- (a) `List<?> listIntersection(List<?> l, Set<?> s)`
- (b) `List<Object> listIntersection(List<Object> l, Set<?> s)`
- (c) `<T> List<T> listIntersection(List<T> l, Set<? extends T> s)`
- (d) `<T> List<T> listIntersection(List<T> l, Set<?> s)`
- (e) `<S,T> List<T> listIntersection(List<T> l, Set<S> s)`

38. (10 punti, 24 giugno 2015) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Nel pattern `Decorator`, l'oggetto decorato è sotto-tipo dell'oggetto decoratore.
- ☐ ☐ Il pattern `Iterator` consente di esaminare una collezione senza esporre la sua struttura interna.
- ☐ ☐ L'interfaccia `Comparator` rappresenta un'istanza del pattern `Template Method`.
- ☐ ☐ Lo scopo del pattern `Composite` è di aggiungere funzionalità ad una data classe.
- ☐ ☐ Nel pattern `Factory Method`, i prodotti concreti sono sotto-tipi del prodotto generico.

39. (25 punti, 8 luglio 2015) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, long l, float m) { return "A1"; }
    public String f(A x, byte l, int m) { return "A2"; }
    public String f(B x, short l, boolean m) { return "A3"; }
}
class B extends A {
    public String f(A y, long m, float p) { return "B1"; }
    public String f(A y, long m, long p) { return "B2"; }
    public String f(Object y, double m, float p) { return "B3"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;

        System.out.println(alfa.f(alfa, (short)500, 1));
        System.out.println(beta.f(alfa, (short)500, 1));
        System.out.println(beta.f(beta, (short)500, 1));
        System.out.println(beta.f(beta, (byte)1, 1));
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

40. **TimeToFinish.** (35 punti, 8 luglio 2015) Implementare la classe thread-safe `TimeToFinish`, che permette a diversi thread di comunicare quanto tempo manca alla propria terminazione. Il metodo `setEstimate` accetta un `long` che rappresenta il numero di millisecondi che mancano al thread chiamante per terminare la sua esecuzione (cioè, il *time-to-finish*). Il metodo `maxTimeToFinish` restituisce *in tempo costante* il numero di millisecondi necessari perché tutti i thread terminino. La stringa restituita da `toString` riporta il *time-to-finish* di tutti i thread.

Suggerimento: il metodo statico `System.currentTimeMillis()` restituisce un `long` che rappresenta il numero di millisecondi trascorsi dal 1 gennaio 1970 (*POSIX time*).

Caso d'uso:

```
TimeToFinish ttf = new TimeToFinish();
ttf.setEstimate(5000);
// a questo punto un altro thread invoca ttf.setEstimate(3000)
Thread.sleep(500);
System.out.println("Tempo_rimanente:_ " + ttf.maxTimeToFinish() + "_millisecondi.");
System.out.println(ttf);
```

Output:

```
Tempo rimanente: 4500 millisecondi.
Thread 1: 2500
Thread main: 4500
```

41. **Question e Answer.** (35 punti, 8 luglio 2015) Per un sito di domande e risposte, realizzare le classi `Question` e `Answer`. Ogni risposta è associata ad un'unica domanda e gli utenti possono votare la risposta migliore invocando il metodo `voteUp` di `Answer`. Inoltre, il metodo `getBestAnswer` restituisce *in tempo costante* la risposta (o una delle risposte) che ha ottenuto il maggior numero di voti.

Rispettare il seguente caso d'uso.

Caso d'uso:

```
Question q = new Question("Dove_si_trova_Albuquerque?");
Answer a1 = new Answer(q, "Canada");
Answer a2 = new Answer(q, "New_Mexico");
a1.voteUp();
System.out.println(q.getBestAnswer());
a2.voteUp();
a2.voteUp();
System.out.println(q.getBestAnswer());
```

Output:

```
Canada
New Mexico
```

42. **SetComparator.** (20 punti, 8 luglio 2015) Valutare le seguenti specifiche per un comparatore tra insiemi, indicando quali sono valide e perché.

Dati due `Set<?>` a e b , `compare(a,b)` restituisce (nei casi non previsti, restituisce zero):

- (a) -1 se $a \cap b = \emptyset$ (a e b disgiunti); 1 se $a \cap b \neq \emptyset$.
 - (b) -1 se $a \subset b$ (a contenuto strettamente in b); 1 se $b \subset a$.
 - (c) -1 se a è vuoto e b no; 1 se a contiene un oggetto che non appartiene a b .
 - (d) -1 se a è un `HashSet` e b è un `TreeSet`; 1 se a è un `TreeSet` e b è un `HashSet`.
43. (10 punti, 8 luglio 2015) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Nel pattern **Observer**, l'osservatore ha un metodo per registrarsi presso un oggetto da osservare.
- ☐ ☐ Aggiungere un tasto (JButton) ad una finestra AWT rappresenta un'applicazione del pattern **Decorator**.
- ☐ ☐ Il pattern **Composite** prevede che un contenitore si possa comportare come un oggetto primitivo.
- ☐ ☐ Il pattern **Factory Method** prevede che una classe costruisca oggetti di un'altra classe.
- ☐ ☐ Il pattern **Strategy** si implementa tipicamente con una classe astratta.

44. (28 punti, 20 gennaio 2015) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, B y) { return "A1"; }
    public String f(A[] x, B y) { return "A2"; }
}
class B extends A {
    public String f(Object x, B y) { return "B1+" + f(y, null); }
    public String f(B[] x, C y) { return "B2"; }
}
class C extends B {
    public String f(A[] x, A y) { return "C1+" + f(null, y); }
    public String f(B[] x, C y) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        B[] beta = new C[10];
        A[] alfa = beta;
        beta[0] = new C();

        System.out.println(beta[0].f(beta, beta[0]));
        System.out.println(beta[0].f(alfa, beta[2]));
        System.out.println(beta[0].f(alfa, alfa[0]));
        System.out.println(6 & 7);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
45. **DataSeries.** (22 punti, 20 gennaio 2015) Realizzare la classe **DataSeries**, che rappresenta una serie storica di dati numerici (ad es., la popolazione di una regione anno per anno). Il metodo `set` imposta il valore della serie per un dato anno. Il metodo statico `comparatorByYear` accetta un anno e restituisce un comparatore tra serie di dati che confronta il valore delle due serie per quell'anno.

Esempio d'uso:

```
DataSeries pop1 = new DataSeries(),
pop2 = new DataSeries();
pop1.set(2000, 15000.0); pop1.set(2005, 18500.0); pop1.set(2010, 19000.0);
pop2.set(2000, 12000.0); pop2.set(2005, 16000.0); pop2.set(2010, 21000.0);
Comparator<DataSeries> c2005 = DataSeries.comparatorByYear(2005),
c2010 = DataSeries.comparatorByYear(2010);
System.out.println(c2005.compare(pop1, pop2));
System.out.println(c2010.compare(pop1, pop2));
```

Output:

1
-1

46. **Relation.** (35 punti, 20 gennaio 2015) Realizzare la classe `Relation`, che rappresenta una relazione binaria tra un insieme `S` e un insieme `T`. In pratica, una `Relation` è analoga ad una `Map`, con la differenza che la `Relation` accetta chiavi duplicate.

Il metodo `put` aggiunge una coppia di oggetti alla relazione. Il metodo `remove` rimuove una coppia di oggetti dalla relazione. Il metodo `image` accetta un oggetto `x` di tipo `S` e restituisce l'insieme degli oggetti di tipo `T` che sono in relazione con `x`. Il metodo `preImage` accetta un oggetto `x` di tipo `T` e restituisce l'insieme degli oggetti di tipo `S` che sono in relazione con `x`.

Esempio d'uso:	Output:
<pre>Relation<Integer,String> r = new Relation<Integer,String>(); r.put(0, "a"); r.put(0, "b"); r.put(0, "c"); r.put(1, "b"); r.put(2, "c"); r.remove(0, "a"); Set<String> set0 = r.image(0); Set<Integer> setb = r.preImage("b"); System.out.println(set0); System.out.println(setb);</pre>	<pre>[b, c] [0, 1]</pre>

47. **difference.** (20 punti, 20 gennaio 2015) Il metodo `difference` accetta due insiemi `a` e `b` e restituisce un nuovo insieme, che contiene gli elementi che appartengono ad `a` ma non a `b` (cioè, la differenza insiemistica tra `a` e `b`).

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporre un'altra.

- (a) `Set<?> difference(Set<?> a, Set<?> b)`
 - (b) `Set<Object> difference(Set<?> a, Set<?> b)`
 - (c) `Set<Object> difference(Set<String> a, Set<String> b)`
 - (d) `<T> Set<T> difference(Set<T> a, Set<?> b)`
 - (e) `<T> Set<T> difference(Set<? extends T> a, Set<? extends T> b)`
 - (f) `<T> Set<T> difference(Set<T> a, Set<? extends T> b)`
48. (20 punti, 20 gennaio 2015) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ In Java ad ogni thread di esecuzione è sempre associato un oggetto `Thread`.
- ☐ ☐ Un thread non può invocare `interrupt` su sé stesso (cioè, sull'oggetto `Thread` che gli corrisponde).
- ☐ ☐ `wait` è un metodo di `Thread`.
- ☐ ☐ Invocare `x.wait()` senza possedere il mutex di `x` provoca un errore di compilazione.
- ☐ ☐ Un campo di classe non può essere `synchronized`.
- ☐ ☐ Il pattern `Composite` prevede che gli oggetti primitivi abbiano un riferimento al contenitore in cui sono inseriti.
- ☐ ☐ Nel framework MVC, ogni oggetto *view* comunica con almeno un oggetto *model*.
- ☐ ☐ La scelta del layout di un container AWT rappresenta un'istanza del pattern `Composite`.
- ☐ ☐ Nel pattern `Factory Method` i client non hanno bisogno di conoscere il tipo effettivo dei prodotti.
- ☐ ☐ Il pattern `Decorator` prevede che l'oggetto da decorare abbia un metodo per aggiungere una decorazione.

49. (33 punti, 3 novembre 2014) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object a, B b) { return "A1"; }
    public String f(C a, B b) { return "A2"; }
}
class B extends A {
    public String f(Object a, B b) { return "B1+" + f(b, null); }
    public String f(A a, B b) { return "B2"; }
}
class C extends B {
    public String f(Object a, B b) { return "C1+" + f(this, b); }
    private String f(B a, B b) { return "C2"; }
}
public class Test0 {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;

        System.out.println(gamma.f(beta, beta));
        System.out.println(alfa.f(beta, gamma));
        System.out.println(9 & 12);
    }
}
```

- Indicare l'output del programma.
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
50. **Pizza.** (50 punti, 3 novembre 2014) [CROWDGRADER] Realizzare la classe `Pizza`, in modo che ad ogni oggetto si possano assegnare degli ingredienti, scelti da un elenco fissato. Ad ogni ingrediente è associato il numero di calorie che apporta alla pizza. Gli oggetti di tipo `Pizza` sono dotati di ordinamento naturale, sulla base del numero totale di calorie. Infine, gli oggetti di tipo `Pizza` sono anche clonabili.

Esempio d'uso:	Output:
<pre>Pizza margherita = new Pizza(), marinara = new Pizza(); margherita.addIngrediente(Pizza.Ingrediente.POMODORO); margherita.addIngrediente(Pizza.Ingrediente.MOZZARELLA); marinara.addIngrediente(Pizza.Ingrediente.POMODORO); marinara.addIngrediente(Pizza.Ingrediente.AGLIO); Pizza altra = margherita.clone(); System.out.println(altra.compareTo(marinara));</pre>	1

51. **CrazyIterator.** (17 punti, 3 novembre 2014) Individuare l'output del seguente programma. Dire se la classe `FunnyIterator` rispetta il contratto dell'interfaccia `Iterator`. In caso negativo, giustificare la risposta.

```
1 public class FunnyIterator implements Iterator {
2     private String msg = "";
3
4     public Object next() {
5         if (msg.equals("")) msg = "ah";
6         else msg += msg;
7         return msg;
8     }
}
```

```

9
10 public boolean hasNext() { return msg.length() < 5; }
11 public void    remove() { }
12 public void    addChar() { msg += "b"; }
13
14 public static void main(String[] args) {
15     Iterator i = new FunnyIterator();
16
17     do {
18         System.out.println(i.next());
19     } while (i.hasNext());
20 }
21 }

```

52. (25 punti, 5 febbraio 2015) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(A x, A y, B z) { return "A1"; }
    public String f(A x, B y, C z) { return "A2"; }
}
class B extends A {
    public String f(Object x, A y, B z) { return "B1"; }
    public String f(B x, B y, B z) { return "B2"; }
}
class C extends B {
    public String f(A x, A y, B z) { return "C1"; }
    public String f(C x, B y, A z) { return "C2" + f(z, y, null); }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;

        System.out.println(beta.f(alfa, beta, beta));
        System.out.println(beta.f(gamma, beta, beta));
        System.out.println(gamma.f(beta, alfa, beta));
        System.out.println(gamma.f(gamma, beta, beta));
        System.out.println(129573 & 129572);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
53. **Box.** (25 punti, 5 febbraio 2015) Realizzare la classe `Box`, che rappresenta una scatola, caratterizzata dalle sue tre dimensioni: altezza, larghezza e profondità. Due scatole sono considerate uguali (da `equals`) se hanno le stesse dimensioni. Le scatole sono dotate di ordinamento naturale basato sul loro volume. Infine, il metodo `fitsIn`, invocato su una scatola x , accetta un'altra scatola y e restituisce `true` se e solo se y è sufficientemente grande da contenere x .

Esempio d'uso:	Output:
<pre> Box grande = new Box(20, 30, 40), grande2 = new Box(30, 20, 40), piccolo = new Box(10, 10, 50); System.out.println(grande.equals(grande2)); System.out.println(grande.compareTo(piccolo)); System.out.println(piccolo.fitsIn(grande)); </pre>	<pre> false 1 false </pre>

54. **ForgetfulSet.** (43 punti, 5 febbraio 2015) Realizzare la classe **ForgetfulSet**, che rappresenta un insieme che “dimentica” progressivamente gli oggetti inseriti. Precisamente, ciascun oggetto inserito viene rimosso automaticamente dopo un ritardo specificato inizialmente come parametro del costruttore. Quindi, il costruttore accetta il ritardo in millisecondi; il metodo **add** inserisce un oggetto nell’insieme; il metodo **contains** accetta un oggetto e restituisce **true** se e solo se quell’oggetto appartiene correntemente all’insieme.

La classe deve utilizzare un numero limitato di thread e deve risultare *thread-safe*.

Suggerimento: il metodo statico `System.currentTimeMillis()` restituisce il numero di millisecondi trascorsi dal 1 gennaio 1970 (*POSIX time*).

Esempio d’uso:

```
ForgetfulSet<String> s = new ForgetfulSet<String>(1000);
s.add("uno");
s.add("due");
System.out.println(s.contains("uno") + ", " + s.contains("due") + ", " + s.contains("tre"));
Thread.sleep(800);
s.add("tre");
System.out.println(s.contains("uno") + ", " + s.contains("due") + ", " + s.contains("tre"));
Thread.sleep(800);
System.out.println(s.contains("uno") + ", " + s.contains("due") + ", " + s.contains("tre"));
```

Output:

```
true, true, false
true, true, true
false, false, true
```

55. **reverseList.** (22 punti, 5 febbraio 2015) Il metodo **reverseList** accetta una lista e restituisce una nuova lista, che contiene gli stessi elementi della prima, ma in ordine inverso.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l’intestazione migliore oppure proporre un’altra.

- (a) `List<?> reverseList(List<?> l)`
- (b) `<T> List<? extends T> reverseList(List<? super T> l)`
- (c) `<T extends List<?>> T reverseList(T l)`
- (d) `<T> List<T> reverseList(List<T> l)`
- (e) `List<Object> reverseList(List<Object> l)`

56. (10 punti, 5 febbraio 2015) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l’esercizio vale 0.

Vero Falso

- ☐ ☐ Tipicamente l’aggregazione tra decoratore e oggetto decorato viene stabilita da un costruttore.
- ☐ ☐ Il pattern **Strategy** consente ai client di fornire versioni particolari di un algoritmo.
- ☐ ☐ Le interfacce **Iterator** e **Iterable** rappresentano un’istanza del pattern **Factory Method**.
- ☐ ☐ Il pattern **Template Method** prevede che un metodo concreto di una classe ne invochi uno astratto della stessa classe.
- ☐ ☐ L’interfaccia **Collection** è un’istanza del pattern **Composite**.

57. **VoteBox.** (30 punti, 7 febbraio 2011) Si implementi la classe `VoteBox`, che rappresenta un'urna elettorale, tramite la quale diversi thread possono votare tra due alternative, rappresentate dai due valori booleani. Il costruttore accetta il numero totale n di thread aventi diritto al voto. La votazione termina quando n thread diversi hanno votato. In caso di pareggio, vince il valore `false`.

Il metodo `vote`, con parametro boolean e nessun valore di ritorno, permette ad un thread di votare, e solleva un'eccezione se lo stesso thread tenta di votare più di una volta. Il metodo `waitForResult`, senza argomenti e con valore di ritorno booleano, restituisce il risultato della votazione, mettendo il thread corrente in attesa se la votazione non è ancora terminata. Infine, il metodo `isDone` restituisce `true` se la votazione è terminata, e `false` altrimenti.

E' necessario evitare eventuali *race condition*.

Esempio d'uso:	Output dell'esempio d'uso:
<pre>VoteBox b = new VoteBox(10); b.vote(true); System.out.println(b.isDone()); b.vote(false);</pre>	<pre>false Exception in thread "main"...</pre>

58. (26 punti, 7 febbraio 2011) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(int x, double y) { return f(x, x); }
    public String f(double x, double y) { return "A2"; }
}
class B extends A {
    public String f(long x, double y) { return f(x, x); }
    private String f(long x, long y) { return "B2"; }
    public String f(int x, double y) { return "B3"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(1, 2));
        System.out.println(beta.f(1.0, 2));
        System.out.println((2 == 2) && (null instanceof Object));
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (10 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (Attenzione: le chiamate in questo esercizio sono quattro) (16 punti)

59. **Operai.** (25 punti, 7 febbraio 2011) Un operaio (classe `Op`) è caratterizzato da nome (`String`) e salario (`short`), mentre un operaio specializzato (classe `OpSp`, sottoclasse di `Op`), in aggiunta possiede una specializzazione (riferimento ad un oggetto di tipo `Specialty`). Dire quali dei seguenti sono criteri validi di uguaglianza (`equals`) tra operai e operai specializzati, giustificando la risposta. Implementare comunque il criterio (a), indicando chiaramente in quale/i classe/i va ridefinito il metodo `equals`.

- (a) Due operai semplici sono uguali se hanno lo stesso nome. Due operai specializzati, in più, devono avere anche la stessa specializzazione (nel senso di `==`). Un operaio semplice non è mai uguale ad un operaio specializzato.

- (b) Come il criterio (a), tranne che un operaio semplice è uguale ad un operaio specializzato se hanno lo stesso nome e la specializzazione di quest'ultimo è null.
- (c) Due operai di qualunque tipo sono uguali se hanno lo stesso salario.
- (d) Gli operai semplici sono tutti uguali tra loro. Ciascun operaio specializzato è uguale solo a se stesso.

60. **MakeMap.** (24 punti, 7 febbraio 2011) Scrivere un metodo che accetta due liste (**List**) k e v di pari lunghezza, e restituisce una mappa in cui all'elemento i -esimo di k viene associato come valore l'elemento i -esimo di v .

Il metodo lancia un'eccezione se le liste non sono di pari lunghezza, oppure se k contiene elementi duplicati.

Si ricordi che non è opportuno utilizzare l'accesso posizionale su liste generiche.

61. (20 punti, 7 febbraio 2011) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ `Collection<?>` è supertipo di `Set<Integer>`.
- ☐ ☐ `TreeMap` è un'interfaccia.
- ☐ ☐ Tutte le classi implementano automaticamente `Cloneable`.
- ☐ ☐ `getClass` è un metodo pubblico di `Object`.
- ☐ ☐ `Enum` è una classe parametrica.
- ☐ ☐ Di un metodo `final` non si può fare l'overloading.
- ☐ ☐ Nel pattern `Decorator`, l'oggetto decorato e quello decoratore implementano una stessa interfaccia.
- ☐ ☐ Le interfacce `Iterator` e `Iterable` sono un esempio del pattern `Template Method`.
- ☐ ☐ Nel framework `Model-View-Controller`, gli oggetti `Model` sono indipendenti dall'interfaccia utente utilizzata.
- ☐ ☐ Il pattern `Composite` prevede che il tipo effettivo degli oggetti contenuti sia sottotipo del tipo effettivo dell'oggetto contenitore.

62. **Time.** (30 punti, 14 settembre 2010) Implementare la classe `Time`, che rappresenta un orario della giornata (dalle 00:00:00 alle 23:59:59). Gli orari devono essere confrontabili secondo `Comparable`. Il metodo `minus` accetta un altro orario x come argomento e restituisce la differenza tra questo orario e x , sotto forma di un nuovo oggetto `Time`. La classe fornisce anche gli orari predefiniti `MIDDAY` e `MIDNIGHT`.

Esempio d'uso:	Output dell'esempio d'uso:
<pre>Time t1 = new Time(14,35,0); Time t2 = new Time(7,10,30); Time t3 = t1.minus(t2); System.out.println(t3); System.out.println(t3.compareTo(t2)); System.out.println(t3.compareTo(Time.MIDDAY));</pre>	<pre>7:24:30 1 -1</pre>

63. (24 punti, 14 settembre 2010) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(A x, A[] arr) { return "A1"; }
    public String f(Object x, B y) { return "A2"; }
}
class B extends A {
    public String f(B x, B y) { return "B1"; }
}
class C extends B {
    public String f(A x, A[] arr) { return "C1"; }
    public String f(B x, C y) { return "C2"; }
    private String f(C x, C y) { return "C3"; }
}
public class Test {
    public static void main(String[] args) {
        A[] arr = new B[10];
        C gamma = new C();
        B beta = gamma;
        A alfa = null;
        System.out.println(beta.f(gamma,gamma));
        System.out.println(beta.f(beta,arr));
        System.out.println(gamma.f(beta,alfa));
        System.out.println(gamma.f(alfa,beta));
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (12 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)

64. **Intersect.** (16 punti, 14 settembre 2010) Implementare il metodo statico `intersect`, che accetta come argomenti due `Collection` x e y e restituisce una nuova `Collection` che contiene l'intersezione di x ed y (cioè, gli oggetti comuni ad entrambe le collezioni).

Prestare particolare attenzione alla scelta della firma del metodo.

65. **ExecuteInParallel.** (35 punti, 14 settembre 2010) Implementare il metodo statico `executeInParallel`, che accetta come argomenti un array di `Runnable` e un numero naturale k , ed esegue tutti i `Runnable` dell'array, k alla volta.

In altre parole, all'inizio il metodo fa partire i primi k `Runnable` dell'array. Poi, non appena uno dei `Runnable` in esecuzione termina, il metodo ne fa partire un altro, preso dall'array, fino ad esaurire tutto l'array.

Sarà valutato negativamente l'uso di attesa attiva.

66. (20 punti, 14 settembre 2010) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Un campo statico è condiviso da tutti gli oggetti della sua classe.
- ☐ ☐ `RandomAccess` è una interfaccia parametrica.
- ☐ ☐ Gli oggetti di tipo `Integer` sono immutabili.
- ☐ ☐ Le enumerazioni sono sempre clonabili.
- ☐ ☐ Un file sorgente Java può contenere più classi.
- ☐ ☐ I design pattern offrono una casistica dei più comuni errori di progettazione.
- ☐ ☐ Il pattern `Strategy` suggerisce di utilizzare un oggetto per rappresentare un algoritmo.

- ☐ ☐ Il metodo `toString` della classe `Object` rappresenta un esempio del pattern `Factory Method`.
- ☐ ☐ Il pattern `Observer` si applica quando un oggetto genera eventi destinati ad altri oggetti.
- ☐ ☐ Il pattern `Composite` e il pattern `Decorator` sono soluzioni alternative allo stesso problema.

67. **Segment.** (34 punti, 30 novembre 2010) Implementare la classe `Segment`, che rappresenta un segmento collocato nel piano cartesiano. Il costruttore accetta le coordinate dei due vertici, nell'ordine x_1, y_1, x_2, y_2 . Il metodo `getDistance` restituisce la distanza tra la retta che contiene il segmento e l'origine del piano. Ridefinire il metodo `equals` in modo che due segmenti siano considerati uguali se hanno gli stessi vertici. Fare in modo che i segmenti siano clonabili.

Si ricordi che:

- L'area del triangolo con vertici di coordinate (x_1, y_1) , (x_2, y_2) e (x_3, y_3) è data da:

$$\frac{|x_1(y_2 - y_3) - x_2(y_1 - y_3) + x_3(y_1 - y_2)|}{2}.$$

Esempio d'uso:	Output dell'esempio d'uso:
<pre>Segment s1 = new Segment(0.0, -3.0, 4.0, 0.0); Segment s2 = new Segment(4.0, 0.0, 0.0, -3.0); Segment s3 = s2.clone(); System.out.println(s1.equals(s2)); System.out.println(s1.getDistance());</pre>	<pre>true 2.4</pre>

68. (26 punti, 30 novembre 2010) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(B x, Object y) { return f(x, x); }
    public String f(B x, A y) { return "A2"; }
}
class B extends A {
    public String f(B x, Object y) { return f(x, x); }
    private String f(B x, B y) { return "B2"; }
    public String f(B x, A y) { return "B3"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(beta, "ciao"));
        System.out.println(beta.f(beta, new A[10]));
        System.out.println((1 == 2) || (7 >= 7));
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (10 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (Attenzione: le chiamate in questo esercizio sono quattro) (16 punti)

69. (26 punti, 30 novembre 2010) Con riferimento all'esercizio 1, determinare quali tra i seguenti criteri sono validi per un `Comparator` tra segmenti, e perché. (18 punti)

Inoltre, implementare uno dei tre criteri. (8 punti)

Dati due `Segment` `a` e `b`, `compare(a,b)` deve restituire:

- (a)
 - -1, se a è più corto di b ;
 - 1, se b è più corto di a ;
 - 0, altrimenti.
- (b)
 - -1, se a , considerato come insieme di punti, è contenuto in b ;
 - 1, se a non è contenuto in b , ma b è contenuto in a ;
 - 0, altrimenti
- (c)
 - -1, se a è orizzontale e b è verticale;
 - 1, se b è orizzontale e a è verticale;
 - 0, altrimenti.

70. **SelectKeys.** (19 punti, 30 novembre 2010) Scrivere un metodo che accetta una lista l e una mappa m , e restituisce una nuova lista che contiene gli elementi di l che compaiono come chiavi in m . Porre particolare attenzione alla scelta della firma.
71. (20 punti, 30 novembre 2010) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ L'interfaccia `List<E>` estende `Iterator<E>`.
- ☐ ☐ `Collection<?>` è supertipo di `Set<Integer>`.
- ☐ ☐ Un metodo statico può essere astratto.
- ☐ ☐ `notify` è un metodo pubblico di `Object`.
- ☐ ☐ `start` è un metodo statico di `Thread`.
- ☐ ☐ L'invocazione `x.join()` mette il thread corrente in attesa che il thread `x` termini.
- ☐ ☐ Nel pattern `Decorator`, l'oggetto decorato e quello decoratore implementano una stessa interfaccia.
- ☐ ☐ Le interfacce `Iterator` e `Iterable` sono un esempio del pattern `Composite`.
- ☐ ☐ Il pattern `Template Method` si applica quando un algoritmo si basa su determinate operazioni primitive.
- ☐ ☐ Il pattern `Observer` permette a diversi oggetti di ricevere gli eventi generati da un altro oggetto.

72. **QueueOfTasks.** (25 punti, 28 giugno 2010) Implementare la classe `QueueOfTasks`, che rappresenta una sequenza di azioni da compiere, ciascuna delle quali rappresentata da un oggetto `Runnable`. Il metodo `add` aggiunge un'azione alla sequenza. Le azioni vengono eseguite nell'ordine in cui sono state passate ad `add` (FIFO), una dopo l'altra, automaticamente.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> QueueOfTasks q = new QueueOfTasks(); Runnable r1 = new Runnable() { public void run() { try { Thread.sleep(2000); } catch (InterruptedException e) { return; } System.out.println("Io_adoro_i_thread!"); } }; Runnable r2 = new Runnable() { public void run() { System.out.println("Io_odio_i_thread!"); } }; q.add(r1); q.add(r1); q.add(r2); System.out.println("fatto."); </pre>	<pre> fatto. Io adoro i thread! (dopo 2 secondi) Io adoro i thread! (dopo 2 secondi) Io odio i thread! </pre>

73. (25 punti, 28 giugno 2010) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(A x, A[] arr) { return "A1"; }
    public String f(Object x, Object y) { return "A2"; }
}
class B extends A {
    public String f(B x, Object[] y) { return "B1"; }
}
class C extends B {
    public String f(A x, A[] arr) { return "C1"; }
    public String f(B x, Object y) { return "C2"; }
    public String f(C x, B y) { return "C3"; }
}
public class Test {
    public static void main(String[] args) {
        A[] arr = new B[10];
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(beta.f(gamma,arr));
        System.out.println(gamma.f(arr,alfa));
        System.out.println(gamma.f(beta,alfa));
        System.out.println(5 | 7);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (16 punti)
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (9 punti)
74. **PartiallyComparable.** (30 punti, 28 giugno 2010) L'interfaccia `PartComparable` (per *partially comparable*) rappresenta un tipo i cui elementi sono *parzialmente* ordinati.

```

public interface PartComparable<T> {
    public PartComparison compareTo(T x);
}

```

```

}
public enum PartComparison {
    SMALLER, EQUAL, GREATER, UNCOMP;
}

```

Implementare la classe **POSet** (per *partially ordered set*), che rappresenta un insieme parzialmente ordinato, i cui elementi implementano l'interfaccia **PartComparable**. Un oggetto di questo insieme è detto *massimale* se nessun altro oggetto nell'insieme è maggiore di lui.

Il metodo **add** aggiunge un oggetto all'insieme, mentre il metodo **isMaximal** restituisce vero se l'oggetto passato come argomento è uno degli oggetti massimali dell'insieme, restituisce falso se tale oggetto appartiene all'insieme, ma non è massimale, ed infine solleva un'eccezione se l'oggetto non appartiene all'insieme. Il metodo **isMaximal** deve terminare in tempo costante.

<pre> // Stringhe, ordinate parzialmente dalla relazione di prefisso class POString implements PartComparable<POString> { ... } POSet<POString> set = new POSet<POString>(); set.add(new POString("architetto")); set.add(new POString("archimede")); set.add(new POString("archi")); set.add(new POString("bar")); set.add(new POString("bari")); System.out.println(set.isMaximal(new POString("archimede"))); ; System.out.println(set.isMaximal(new POString("bar"))); set.add(new POString("archimedeo")); System.out.println(set.isMaximal(new POString("archimede"))); ; </pre>	<p>Output dell'esempio d'uso:</p> <pre> true false false </pre>
---	---

75. (20 punti, 28 giugno 2010) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ L'interfaccia **Map<K,V>** estende **Iterable<K>**.
- ☐ ☐ **Iterator<?>** è supertipo di **Iterator<? extends Employee>**.
- ☐ ☐ **RandomAccess** è una interfaccia vuota.
- ☐ ☐ Una classe astratta può estendere un'altra classe astratta.
- ☐ ☐ Una classe interna può avere visibilità **protected**.
- ☐ ☐ Una classe anonima non può avere costruttore.
- ☐ ☐ Nel pattern **Decorator**, non è necessario che l'oggetto decorato sia consapevole della decorazione.
- ☐ ☐ Nel framework MVC, le classi "model" si occupano di presentare i dati all'utente.
- ☐ ☐ Il pattern **Strategy** si applica quando un algoritmo si basa su determinate operazioni primitive.
- ☐ ☐ Il pattern **Factory Method** permette ad una gerarchia di produttori di produrre una gerarchia di prodotti.

76. **MultiProgressBar**. (35 punti, 4 marzo 2011) Si supponga che una applicazione divida un'operazione complessa tra più thread, che procedono in parallelo. Si implementi la classe **MultiProgressBar**, di cui ciascun oggetto serve a tenere traccia dello stato di avanzamento dei thread coinvolti in una data operazione.

Il costruttore accetta il numero totale n di thread coinvolti. Il metodo `progress`, con un argomento intero e senza valore di ritorno, consente ad un thread di dichiarare il suo stato di avanzamento, in percentuale. Tale metodo lancia un'eccezione se lo stesso thread dichiara uno stato di avanzamento inferiore ad uno precedentemente dichiarato. Il metodo `getProgress`, senza argomenti e con valore di ritorno intero, restituisce il *minimo* stato di avanzamento tra tutti i thread coinvolti.

E' necessario evitare eventuali *race condition*.

Un esempio d'uso verrà fornito alla lavagna.

77. (26 punti, 4 marzo 2011) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, B y)      { return "A1"; }
    public String f(C x, Object y) { return "A2"; }
}
class B extends A {
    public String f(A x, B y) { return f(x, x); }
    private String f(A x, A y) { return "B2"; }
    public String f(C x, B y) { return "B3"; }
}
class C extends B {
    public String f(C x, B y) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = new C();
        System.out.println(alfa.f(alfa, alfa));
        System.out.println(alfa.f(alfa, beta));
        System.out.println(alfa.f((C)alfa, beta));
        System.out.println(alfa.getClass() == C.class);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (10 punti)
 - Per ogni chiamata ad un metodo *user-defined*, indicare la lista delle firme candidate. (Attenzione: le chiamate in questo esercizio sono quattro) (16 punti)
78. (20 punti, 4 marzo 2011) La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A<T extends Cloneable> extends B<T> {
    private Cloneable t, u;
    private B<String> b;
    private int i;

    public A(T x) {
        t = x;
        u = g1();
        b = g2(x);
        i = this.compareTo("ciao");
    }

    public Double f(Object o) {
        Number n = super.f(o);
        if (n instanceof Double) return (Double)n;
        else return null;
    }
}
```

79. **PrintBytes.** (24 punti, 4 marzo 2011) Scrivere un metodo statico `printBytes`, che prende come argomento un `long` che rappresenta un numero di byte minore di 10^{15} , e restituisce una stringa in cui il numero di byte viene riportato nell'unità di misura più comoda per la lettura, tra: bytes, kB, MB, GB, e TB. Più precisamente, il metodo deve individuare l'unità che permetta di esprimere (approssimativamente) il numero di byte dato utilizzando tre cifre intere e una frazionaria. L'approssimazione può essere per troncamento oppure per arrotondamento.

input	output
123	"123 bytes"
3000	"3.0 kB"
19199	"19.1 kB"
12500000	"12.5 MB"
710280000	"710.2 MB"
72000538000	"72.0 GB"

80. (20 punti, 4 marzo 2011) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ `Iterable<?>` è supertipo di `Iterator<Integer>`.
 - ☐ ☐ `Runnable` è un'interfaccia.
 - ☐ ☐ `clone` è un metodo pubblico di `Object`.
 - ☐ ☐ Le variabili locali possono essere `final`.
 - ☐ ☐ `wait` è un metodo di `Thread`.
 - ☐ ☐ Una classe interna statica può avere campi istanza (attributi).
 - ☐ ☐ Nel pattern `Decorator`, l'oggetto decorato mantiene un riferimento a quello decoratore.
 - ☐ ☐ Nel pattern `Template Method`, una classe ha un metodo concreto che chiama metodi astratti.
 - ☐ ☐ Nel pattern `Factory Method`, il prodotto generico dovrebbe essere sottotipo del produttore generico.
 - ☐ ☐ La scelta del layout di un componente grafico è un esempio di applicazione del pattern `Template Method`.
81. **Crosswords.** (45 punti, 3 maggio 2010) Si implementi la classe `Crosswords`, che rappresenta uno schema di parole crociate, inizialmente vuoto. Il costruttore accetta le dimensioni dello schema. Il metodo `addWord` aggiunge una parola allo schema e restituisce `true`, a patto che la parola sia *compatibile* con quelle precedentemente inserite; altrimenti, restituisce `false` senza modificare lo schema. Il metodo prende come argomenti le coordinate iniziali della parola, la parola stessa e la direzione (H per orizzontale e V per verticale).

Le regole di *compatibilità* sono:

- Una parola non si può sovrapporre ad un'altra della stessa direzione.
- Una parola si può incrociare con un'altra solo su di una lettera comune.
- Ogni parola deve essere preceduta e seguita da un bordo o da una casella vuota.

Non è necessario implementare il metodo `toString`. E' opportuno implementare le direzioni H e V in modo che siano le uniche istanze del loro tipo.

Suggerimenti:

- Per evitare di scrivere separatamente i due casi per orizzontale e verticale, è possibile aggiungere i metodi `getChar/setChar`, che prendono come argomenti una riga r , una colonna c , una direzione d (H o V) e un *offset* x , e leggono o scrivono il carattere situato a distanza x dalla casella r, c , in direzione d .

- Il metodo `s.charAt(i)` restituisce il carattere i -esimo della stringa `s` (per i compreso tra 0 e `s.length()-1`).

Esempio d'uso:	Output dell'esempio d'uso:
<pre>Crosswords c = new Crosswords(6, 8); System.out.println(c.addWord(0,3, "casa", Crosswords.V)); System.out.println(c.addWord(2,1, "naso", Crosswords.H)); System.out.println(c.addWord(2,0, "pippo", Crosswords.H)); System.out.println(c);</pre>	<pre>true true false c a *naso* a *</pre>

82. (25 punti, 3 maggio 2010) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    String f(A x, A y) { return "A1"; }
    String f(Object x, B y) { return "A2"; }
}
class B extends A {
    public String f(A x, A y) { return "B1"; }
}
class C extends B {
    public String f(B x, B y) { return "C1"; }
    public String f(B x, Object y) { return "C2"; }
    private String f(C x, B y) { return "C3"; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(beta, alfa));
        System.out.println(beta.f(beta, gamma));
        System.out.println(gamma.f(null, gamma));
        System.out.println(1e100 + 1);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (16 punti)
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (9 punti)
83. **Rebus.** (30 punti, 3 maggio 2010) In enigmistica, un rebus è un disegno dal quale bisogna ricostruire una frase. La traccia del rebus comprende anche la lunghezza di ciascuna parola della soluzione. Dato il seguente scheletro per la classe `Rebus`, dove `picture_name` è il nome del file che contiene il disegno (sempre diverso da `null`) e `word_length` è una lista di interi, che rappresentano la lunghezza di ciascuna parola nella soluzione,

```
class Rebus {
    private String picture_name;
    private List word_length;
}
```

considerare le seguenti specifiche alternative per un `Comparator` tra oggetti `Rebus`. Dati due `Rebus` `a` e `b`, `compare` deve restituire:

- (a)
- -1, se a e b hanno immagini diverse e la lunghezza totale della soluzione di a è minore di quella di b ;
 - 0, se a e b hanno la stessa immagine;
 - 1, altrimenti.
- (b)
- -1, se a e b hanno immagini diverse e la soluzione di b contiene una parola più lunga di tutte le parole di a ;
 - 0, se a e b hanno la stessa immagine, oppure le parole più lunghe delle due soluzioni hanno la stessa lunghezza;
 - 1, altrimenti.
- (c)
- -1, se nessuna delle due immagini ha estensione `png`, e la soluzione di a contiene meno parole di quella di b ;
 - 0, se almeno una delle due immagini ha estensione `png`;
 - 1, altrimenti.

Dire se ciascun criterio di ordinamento è valido, giustificando la risposta. (15 punti)

Implementare il criterio (b). (15 punti)

84. **Tetris.** (25 punti, 26 luglio 2010) Il Tetris è un videogioco il cui scopo è incastrare tra loro pezzi bidimensionali di 7 forme predefinite, all'interno di uno schema rettangolare. Implementare la classe astratta `Piece`, che rappresenta un generico pezzo, e le sottoclassi concrete `T` ed `L`, che rappresentano i pezzi dalla forma omonima.

La classe `Piece` deve offrire i metodi `put`, che aggiunge questo pezzo alle coordinate date di un dato schema, e il metodo `rotate`, che ruota il pezzo di 90 gradi in senso orario (senza modificare alcuno schema). Il metodo `put` deve lanciare un'eccezione se non c'è posto per questo pezzo alle coordinate date. Uno schema viene rappresentato da un array bidimensionale di valori booleani (`false` per libero, `true` per occupato).

E' opportuno raccogliere quante più funzionalità è possibile all'interno della classe `Piece`. Il seguente caso d'uso assume che `print_board` sia un opportuno metodo per stampare uno schema.

Esempio d'uso:	Output dell'esempio d'uso:
<pre>boolean board[][] = new boolean[5][12]; Piece p1 = new T(); p1.put(board, 0, 0); Piece p2 = new L(); p2.put(board, 0, 4); print_board(board); p2.rotate(); p2.put(board, 2, 7); print_board(board);</pre>	<pre>----- X X XXX X XX ----- X X XXX X XX XXX X</pre>

85. (27 punti, 26 luglio 2010) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, int y) { return "A1"; }
    public String f(Object x, double y) { return "A2"; }
}
class B extends A {
    public String f(A x, int y) { return "B1"; }
}
class C extends B {
    public String f(B x, float y) { return "C1"; }
    public String f(Object x, double y) { return "C2"; }
```

```

    public String f(C x, int y) { return "C3:" + f(x, y * 2.0); }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(gamma, 3));
        System.out.println(gamma.f(null, 4));
        System.out.println(gamma.f(beta, 3));
        System.out.println("1" + 1);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (15 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)

86. **TetrisPiece.** (33 punti, 26 luglio 2010) Implementare l'enumerazione `Piece`, che rappresenta i possibili pezzi del Tetris, con almeno le costanti `T` ed `L`, che rappresentano i pezzi dalla forma omonima. Il metodo `put` mette questo pezzo alle coordinate date di uno schema dato, con un dato orientamento. L'orientamento viene fornito dal chiamante tramite un intero compreso tra 0 (pezzo diritto) e 3 (pezzo ruotato di 270 gradi). Lo schema in cui sistemare il pezzo viene rappresentato da un array bidimensionale di valori booleani (`false` per libero, `true` per occupato). Il metodo `put` deve lanciare un'eccezione se non c'è posto per questo pezzo alle coordinate date. Il seguente caso d'uso assume che `print_board` sia un opportuno metodo per stampare uno schema.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> boolean board[][] = new boolean[5][12]; Piece p1 = Piece.T; Piece p2 = Piece.L; p1.put(board, 0, 0, 0); p2.put(board, 0, 4, 0); p2.put(board, 1, 7, 2); print_board(board); </pre>	<pre> ----- X X XXX X XX XX X X </pre>

87. (20 punti, 26 luglio 2010) La seguente classe `A` fa riferimento ad una classe `B`. Implementare la classe `B` in modo che venga compilata correttamente e permetta la compilazione della classe `A`.

```

public class A {
    private B<Integer> b1 = new B<Integer>(null);
    private B<?> b2 = B.f(3);
    private Comparable<? extends Number> c = new B<Double>();

    public Object f() {
        Integer x = b1.getIt();
        Integer y = x + b2.getIt();
        return new B<String>(new A());
    }
}

```

88. (20 punti, 26 luglio 2010) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ `interrupt` è un metodo della classe `Object`.
- ☐ ☐ `Iterator<?>` è supertipo di `Iterator<? extends Employee>`.
- ☐ ☐ `Runnable` è una interfaccia vuota.
- ☐ ☐ Un metodo statico può essere sincronizzato (`synchronized`).
- ☐ ☐ Il metodo `notify` risveglia uno dei thread in attesa su questo oggetto.
- ☐ ☐ Il metodo `notify` può lanciare l'eccezione `InterruptedException`.
- ☐ ☐ Nel pattern `Decorator`, ogni oggetto decoratore contiene un riferimento all'oggetto decorato.
- ☐ ☐ Il pattern `Iterator` consente ad un oggetto di contenerne altri.
- ☐ ☐ Il pattern `Composite` consente di aggiungere funzionalità ad una classe.
- ☐ ☐ Nel pattern `Factory Method`, i produttori concreti sono tutti sotto-tipi del produttore generico.

89. (25 punti, 18 settembre 2014) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, short n) { return "A1"; }
    public String f(A x, int n)      { return "A2"; }
    private String f(B x, double n)  { return "A3"; }
}
class B extends A {
    public String f(A x, double n) { return "B1:" + f(x, (int)n); }
    public String f(B x, double n) { return "B2"; }
    public String f(A x, int n)    { return "B3"; }
}
class C extends B {
    public String f(A x, int n) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta  = new B();
        A alfa  = gamma;
        System.out.println(alfa.f(gamma, (byte)2));
        System.out.println(beta.f(beta, 5.0));
        System.out.println(gamma.f(alfa, (float)5));
        System.out.println(11 | 3);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

90. **EmployeeComparator.** (25 punti, 18 settembre 2014) Un `employee` è caratterizzato da nome (stringa) e salario (intero non negativo). Dire quali delle seguenti sono specifiche valide per un `Comparator` tra `employee`. In caso negativo, motivare la risposta con un controesempio. Nei casi non previsti dalle specifiche, il comparatore restituisce 0.

`compare(x,y)` restituisce:

- (a) -1 se il nome di `x` è uguale a quello di `y`, ma i salari sono diversi; 1 se il salario di `x` è uguale a quello di `y`, ma i nomi sono diversi.
- (b) -1 se il salario di `x` è zero e quello di `y` è maggiore di zero; 1 se il salario di `y` è zero e quello di `x` è maggiore di zero.

- (c) -1 se il salario di x è *maggiore* di quello di y; 1 se il salario di x è *minore* di quello di y.
- (d) -1 se il nome di x precede alfabeticamente quello di y ed il salario di x è minore di quello di y; 1 se il nome di y precede alfabeticamente quello di x.
- (e) -1 se il salario di x è minore di 2000 e quello di y è maggiore di 1000; 1 se il salario di y è minore di 2000 e quello di x è maggiore di 1000.

91. **Contest.** (30 punti, 18 settembre 2014) Un oggetto di tipo **Contest** consente ai client di votare per uno degli oggetti che partecipano a un “concorso”. Implementare la classe parametrica **Contest** con i seguenti metodi: il metodo **add** consente di aggiungere un oggetto al concorso; il metodo **vote** permette di votare per un oggetto; se l’oggetto passato a **vote** non partecipa al concorso (cioè non è stato aggiunto con **add**), viene lanciata un’eccezione; il metodo **winner** restituisce uno degli oggetti che fino a quel momento ha ottenuto più voti.

Tutti i metodi devono funzionare in tempo costante.

Esempio d’uso:	Output:
<pre>Contest<String> c = new Contest<String>(); String r = "Red", b = "Blue", g = "Green"; c.add(r); c.vote(r); c.add(b); c.add(g); c.vote(r); c.vote(b); System.out.println(c.winner());</pre>	Red

92. **atLeastOne.** (35 punti, 18 settembre 2014) Implementare il metodo statico **atLeastOne**, che accetta come argomenti un intero positivo n e un **Runnable** r ed esegue in parallelo n copie di r . Appena una delle copie termina, le altre vengono interrotte (con **interrupt**) e il metodo restituisce il controllo al chiamante.

93. (10 punti, 18 settembre 2014) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l’esercizio vale 0.

Vero Falso

- ☐ ☐ Il metodo **hashCode** di **Object** rappresenta un’istanza del pattern **Template Method**.
- ☐ ☐ Il pattern **Composite** prevede che si possa iterare sui componenti di un oggetto composto.
- ☐ ☐ Il pattern **Factory Method** consente a diverse sottoclassi di creare prodotti diversi.
- ☐ ☐ Nel pattern **Factory Method** il prodotto generico è sottotipo del produttore generico.
- ☐ ☐ Uno dei pre-requisiti del pattern **Iterator** è che più client debbano poter accedere contemporaneamente all’aggregato.

94. (25 punti, 28 novembre 2014) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(C x, short n) { return "A1"; }
    public String f(A x, int n)   { return "A2"; }
    String f(B x, double n) { return "A3"; }
}
class B extends A {
    public String f(A x, double n) { return "B1:" + f(x, (int)n); }
    public String f(B x, double n) { return "B2"; }
    public String f(A x, int n)    { return "B3"; }
}
```

```

class C extends B {
    public String f(A x, int n)    { return "C1"; }
    public String f(B x, double n) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta  = new B();
        A alfa  = gamma;
        System.out.println(alfa.f(beta, (byte)2));
        System.out.println(beta.f(beta, 5.0));
        System.out.println(gamma.f(alfa, (float)5));
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

95. **Coin.** (25 punti, 28 novembre 2014) Realizzare la classe enumerata `Coin`, che rappresenta le 8 monete dell'euro. Il metodo statico `convert` accetta un numero intero n e restituisce una collezione di `Coin` che vale n centesimi.

Nota: per ottenere lo stesso output del caso d'uso, non è necessario ridefinire alcun metodo `toString`.

Esempio d'uso:	Output:
<pre> Collection<Coin> a = Coin.convert(34), b = Coin.convert(296); System.out.println(a); System.out.println(b); </pre>	<pre> [TWENTY, TEN, TWO, TWO] [TWOEuros, FIFTY, TWENTY, TWENTY, FIVE, ONE] </pre>

96. **Alarm.** (25 punti, 28 novembre 2014) Realizzare la classe `Alarm`, nel contesto di un sistema di allarme domestico. Il compito dell'oggetto `Alarm` è di stampare un messaggio se una condizione anomala permane oltre una soglia di tempo prestabilita (un *timeout*). Il costruttore accetta il timeout in secondi. Il metodo `anomalyStart` segnala l'inizio di una situazione anomala, mentre il metodo `anomalyEnd` ne segnala la fine. Se viene invocato `anomalyStart` e poi non viene invocato `anomalyEnd` entro il timeout, l'oggetto `Alarm` produce in output il messaggio "Allarme!".

Se `anomalyStart` viene invocato più volte, senza che sia ancora stato invocato `anomalyEnd`, le invocazioni successive alla prima vengono ignorate (cioè, non azzerano il timeout).

La classe `Alarm` deve risultare *thread safe* e i suoi metodi non devono essere bloccanti.

Esempio d'uso:	Output: (dopo 5 secondi)
<pre> Alarm a = new Alarm(5); a.anomalyStart(); </pre>	<pre> Allarme! </pre>

97. **product.** (15 punti, 28 novembre 2014) Il metodo `product` accetta due insiemi e restituisce il loro prodotto Cartesiano.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporre un'altra.

- (a) `Set<Pair<?,?>> product(Set<?> a, Set<?> b)`

- (b) `<S,T> Set<Pair<S,T>> product(Set<S> a, Set<T> b)`
 (c) `Set<Pair<Object,Object>> product(Set<Object> a, Set<Object> b)`

98. (10 punti, 28 novembre 2014) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Il pattern **Template Method** consente alle sottoclassi di definire versioni concrete di metodi primitivi.
- ☐ ☐ Il pattern **Decorator** consente anche di aggiungere una decorazione ad un oggetto già decorato.
- ☐ ☐ Il pattern **Factory Method** consente a diverse sottoclassi di creare prodotti diversi.
- ☐ ☐ Nel pattern **Factory Method** i produttori concreti sono sottoclassi del produttore generico.
- ☐ ☐ Uno dei pre-requisiti del pattern **Strategy** è che esista un numero predefinito di varianti di un algoritmo.

99. (25 punti, 3 luglio 2014) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, int n) { return "A1"; }
    public String f(A x, int n) { return "A2:" + n; }
    public String f(A x, double n) { return "A3:" + x.f(x, (int) n); }
    private String f(B x, int n) { return "A4"; }
}
class B extends A {
    public String f(A x, double n) { return "B1:" + n; }
    public String f(B x, double n) { return "B2:" + f((A) x, 2); }
    public String f(A x, int n) { return "B3"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(null, 2L));
        System.out.println(beta.f(beta, 5.0));
        System.out.println(12 & 2);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
100. **NutrInfo.** (25 punti, 3 luglio 2014) L'enumerazione **Nutrient** contiene i valori **FAT**, **CARBO** e **PROTEIN**. Implementare la classe **NutrInfo** che rappresenta la scheda nutrizionale di un prodotto gastronomico. Il suo costruttore accetta il numero di chilocalorie. Il metodo `setNutrient` permette di impostare la quantità (in grammi) di ciascun nutriente.

La classe è dotata di un ordinamento naturale, basato sul numero di calorie. Inoltre, la classe offre il metodo statico `comparatorBy` che accetta un valore **Nutrient** e restituisce un comparatore basato sul contenuto di quel nutriente.

Esempio d'uso:

```
NutrInfo x = new NutrInfo(500);
x.setNutrient(Nutrient.FAT, 12.0);
x.setNutrient(Nutrient.CARBO, 20.0);
x.setNutrient(Nutrient.PROTEIN, 15.0);
Comparator<NutrInfo> c = NutrInfo.comparatorBy(Nutrient.FAT);
```

101. **Exchanger.** (30 punti, 3 luglio 2014) Un **Exchanger** è un oggetto che serve a due thread per scambiarsi due oggetti dello stesso tipo. Ciascun thread invocherà il metodo **exchange** passandogli il proprio oggetto e riceverà come risultato l'oggetto passato dall'altro thread. Il primo thread che invoca **exchange** dovrà aspettare che anche il secondo thread invochi quel metodo, prima di ricevere il risultato. Quindi, il metodo **exchange** risulta bloccante per il primo thread che lo invoca e non bloccante per il secondo.

Un **Exchanger** può essere usato per un solo scambio. Ulteriori chiamate ad **exchange** dopo le prime due portano ad un'eccezione.

Nell'esempio d'uso, due thread condividono il seguente oggetto:

```
Exchanger<String> e = new Exchanger<String>();
```

Thread 1:	Thread 2:
String a = e.exchange("ciao"); System.out.println(a);	String a = e.exchange("Pippo"); System.out.println(a);
Output thread 1:	Output thread 2:
Pippo	ciao

102. **subMap.** (35 punti, 3 luglio 2014) Implementare il metodo **subMap** che accetta una mappa e una collezione e restituisce una nuova mappa ottenuta restringendo la prima alle sole chiavi che compaiono nella collezione. Il metodo non modifica i suoi argomenti.

Valutare le seguenti intestazioni per il metodo **subMap**, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporre un'altra.

- (a) `<K> Map<K,?> subMap(Map<K,?> m, Collection<K> c)`
- (b) `<K,V> Map<K,V> subMap(Map<K,V> m, Collection<?> c)`
- (c) `<K,V> Map<K,V> subMap(Map<K,V> m, Collection<? super K> c)`
- (d) `<K,V> Map<K,V> subMap(Map<K,V> m, Collection<? extends K> c)`
- (e) `<K,V> Map<K,V> subMap(Map<K,V> m, Set<K> c)`
- (f) `<K,V,K2 extends K> Map<K,V> subMap(Map<K,V> m, Collection<K2> c)`

103. (10 punti, 3 luglio 2014) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Il pattern **Observer** evita che gli osservatori debbano controllare periodicamente lo stato dell'oggetto osservato (*polling*).
- ☐ ☐ Il pattern **Decorator** prevede un modo per distinguere un oggetto decorato da uno non decorato.
- ☐ ☐ Il pattern **Template Method** si applica in presenza di una gerarchia di classi e sotto-classi.
- ☐ ☐ L'interfaccia **Collection** è un'istanza del pattern **Composite**.

□ □ L'interfaccia `Collection` sfrutta il pattern `Iterator`.

104. (33 punti, 28 aprile 2014) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public static int x = 0;
    public A() { x++; }

    private int f(int a, double b) { return x; }
    public int f(int a, float b) { return x+10; }
    public int f(double a, double b) { return x+20; }
    public String toString() { return f(x, x) + ""; }
}
class B extends A {
    public int f(int a, float b) { return x+30; }
    public int f(int a, int b) { return x+40; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa1 = beta;
        A alfa2 = new A();

        System.out.println(alfa1);
        System.out.println(alfa2);
        System.out.println(beta);
        System.out.println(alfa2.f(2, 3.0));
        System.out.println(beta.f(2, 3));
        System.out.println(beta.f((float) 4, 5));
        System.out.println(15 & 3);
    }
}
```

- Indicare l'output del programma.
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
105. **Shape.** (52 punti, 28 aprile 2014) [CROWDGRADER] Per un programma di geometria piana, realizzare la classe astratta `Shape` e la sottoclasse concreta `Circle`. La classe `Shape` dispone dei metodi `width`, `height`, `posX` e `posY`, che restituiscono rispettivamente la larghezza, l'altezza, la posizione sulle ascisse e la posizione sulle ordinate del più piccolo rettangolo che contiene interamente la figura in questione (le coordinate restituite da `posX` e `posY` si riferiscono all'angolo in basso a sinistra del rettangolo).

Il costruttore di `Circle` accetta le coordinate del centro e il raggio, mentre il metodo `setRadius` consente di modificare il raggio.

Inoltre, le classi offrono le seguenti funzionalità:

- (a) Gli oggetti `Circle` sono uguali secondo `equals` se hanno lo stesso centro e lo stesso raggio.
- (b) Gli oggetti `Shape` sono clonabili.
- (c) Gli oggetti `Shape` sono dotati di ordinamento naturale, sulla base dell'area del rettangolo che contiene la figura.

Esempio d'uso:	Output:
<pre>Shape c1 = new Circle(2.0, 3.0, 1.0); Shape c2 = c1.clone(); System.out.println(c1.posX() + "," + c1.posY()); System.out.println(c1.width() + "," + c1.height()); System.out.println(c1.equals(c2)); ((Circle) c2).setRadius(2.0); System.out.println(c2.posX() + "," + c2.posY());</pre>	<pre>1.0, 2.0 2.0, 2.0 true 0.0, 1.0</pre>

106. **Shape equals.** (15 punti, 28 aprile 2014) Con riferimento all'esercizio 105, dire quali dei seguenti criteri è una valida specifica per l'uguaglianza tra oggetti di tipo `Circle`. In caso negativo, giustificare la risposta con un controesempio.

Due oggetti `Circle` sono uguali se:

- (a) hanno lo stesso centro oppure lo stesso raggio;
- (b) entrambe le circonferenze contengono l'origine all'interno oppure nessuna delle due la contiene;
- (c) il raggio di uno dei due oggetti è maggiore di quello dell'altro.

107. (25 punti, 28 luglio 2014) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, double n) { return "A1"; }
    public String f(A x, int n) { return "A2"; }
    private String f(B x, int n) { return "A3"; }
}
class B extends A {
    private String f(A x, double n) { return "B1"; }
    public String f(B x, double n) { return "B2:" + f((A) x, 2); }
    public String f(A x, long n) { return "B3"; }
}
class C extends B {
    public String f(A x, int n) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;
        System.out.println(alfa.f(gamma, 2L));
        System.out.println(beta.f(beta, 5.0));
        System.out.println(gamma.f(beta, 5.0));
        System.out.println(11 & 3);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

108. **Playlist.** (25 punti, 28 luglio 2014) Implementare le classi `Song` e `Playlist`. Una canzone è caratterizzata dal nome e dalla durata in secondi. Una playlist è una lista di canzoni, compresi eventuali duplicati, ed offre il metodo `add`, che aggiunge una canzone in coda alla lista, e `remove`, che rimuove *tutte* le occorrenze di una canzone dalla lista. Infine, la classe `Playlist` è dotata di ordinamento naturale basato sulla durata totale di ciascuna playlist. Sono preferibili le implementazioni in cui il confronto tra due playlist avvenga in tempo costante.

Esempio d'uso:	Output:
<pre> Song one = new Song("One", 275), two = new Song("Two", 362); Playlist a = new Playlist(), b = new Playlist(); a.add(one); a.add(two); a.add(one); b.add(one); b.add(two); System.out.println(a.compareTo(b)); a.remove(one); System.out.println(a.compareTo(b)); </pre>	<pre> 1 -1 </pre>

109. **PriorityExecutor.** (35 punti, 28 luglio 2014) Un `PriorityExecutor` è un thread che esegue in sequenza una serie di task, in ordine di priorità. Il metodo `addTask` accetta un `Runnable` e una priorità (numero intero) e lo aggiunge ad una coda di task. Quando il `PriorityExecutor` è libero (cioè, non sta eseguendo alcun task), preleva dalla coda *uno dei task con priorità massima* e lo esegue.

Sono preferibili le implementazioni in cui né `addTask` né la ricerca del prossimo task da eseguire richiedano tempo lineare.

Esempio d'uso:	Output:
<pre> Runnable r1 = ..., r2 = ...; PriorityExecutor e = new PriorityExecutor(); e.addTask(r2, 10); e.addTask(r1, 100); e.start(); e.addTask(r2, 15); e.addTask(r1, 50); </pre>	<pre> prima viene eseguito due volte il task r1, poi due volte il task r2 </pre>

110. **inverseMap.** (30 punti, 28 luglio 2014) Implementare il metodo `inverseMap` che accetta una mappa `m` e ne restituisce una nuova, ottenuta invertendo le chiavi con i valori. Se `m` contiene valori duplicati, il metodo lancia un'eccezione. Il metodo non modifica la mappa `m`.

Valutare le seguenti intestazioni per il metodo `inverseMap`, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporre un'altra.

- (a) `<K,V> Map<V,K> inverseMap(Map<?,?> m)`
- (b) `Map<?,?> inverseMap(Map<?,?> m)`
- (c) `<K,V> Map<K,V> inverseMap(Map<V,K> m)`
- (d) `<K,V> Map<K,V> inverseMap(Map<? extends V, ? super K> m)`
- (e) `<K,V> Map<K,V> inverseMap(Map<K,V> m)`
- (f) `<K,V> Map<K,V> inverseMap(Map<? extends V, ? extends K> m)`

111. (10 punti, 28 luglio 2014) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Nel pattern `Composite` sia gli oggetti primitivi che quelli composti hanno un metodo per aggiungere un oggetto alla composizione.
- ☐ ☐ Il metodo `clone` di `Object` è un'istanza del pattern `Template Method`.
- ☐ ☐ Il pattern `Factory Method` consente a diverse sottoclassi di creare prodotti diversi.
- ☐ ☐ Le interfacce `Iterable` e `Iterator` rappresentano un'istanza del pattern `Factory Method`.

- □ Il pattern **Strategy** prevede un'interfaccia (o classe astratta) che rappresenta un algoritmo.

112. (25 punti, 31 gennaio 2014) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, A y) { return "A1"; }
    public String f(A[] x, A y) { return "A2"; }
    public String f(Object[] x, B y) { return "A3"; }
}
class B extends A {
    public String f(A[] x, A y) { return "B1"; }
    public String f(B[] x, A y) { return "B2"; }
}
public class Test {
    public static void main(String[] args) {
        A[] arrA = new B[10];
        B[] arrB = new B[10];
        arrA[0] = arrB[0] = new B();
        System.out.println(arrA[0].f(null, arrA[0]));
        System.out.println(arrB[0].f(arrA, arrB[0]));
        System.out.println(arrB[0].f(arrB, arrA[0]));
        System.out.println("1" + 1);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
113. **Note.** (30 punti, 31 gennaio 2014) Realizzare la classe `BoundedSet`, che rappresenta un insieme di capacità limitata. Il costruttore accetta la capacità massima dell'insieme. La classe deve implementare i metodi `add`, `contains` e `size` secondo il contratto previsto dall'interfaccia `Set`. Se però l'insieme è alla sua capacità massima e si tenta di inserire un nuovo elemento con `add`, prima dell'inserimento sarà cancellato dall'insieme l'elemento che vi è stato inserito prima (cioè, l'elemento più "vecchio" presente nell'insieme).

Fare in modo che sia `add` sia `contains` funzionino in tempo costante.

Esempio d'uso:	Output:
<pre>BoundedSet<Integer> s = new BoundedSet<Integer>(3); s.add(3); s.add(8); s.add(5); s.add(5); System.out.println(s.size()); System.out.println(s.contains(3)); s.add(14); System.out.println(s.size()); System.out.println(s.contains(3));</pre>	<pre>3 true 3 false</pre>

114. **PostOfficeQueue.** (32 punti, 31 gennaio 2014) Implementare una classe `PostOfficeQueue`, che aiuta a gestire la coda in un ufficio postale. Il costruttore accetta il numero totale di sportelli disponibili. Quando l' i -esimo sportello comincia a servire un cliente e quindi diventa occupato, viene invocato (dall'esterno della classe) il metodo `deskStart` passando i come argomento. Quando invece l' i -esimo sportello si libera, viene invocato il metodo `deskFinish` con argomento i . Infine, il metodo `getFreeDesk` restituisce il numero di uno sportello libero. Se non ci sono sportelli liberi, il metodo attende che se ne liberi uno.

Si deve supporre che thread diversi possano invocare concorrentemente i metodi di `PostOfficeQueue`. È necessario evitare *race condition* ed attese attive.

Esempio d'uso:	Output:
PostOfficeQueue poq = new PostOfficeQueue(5);	0
System.out.println(poq.getFreeDesk());	1
poq.deskStart(0);	3
System.out.println(poq.getFreeDesk());	
poq.deskStart(1);	
poq.deskStart(2);	
System.out.println(poq.getFreeDesk());	

115. **isMax.** (28 punti, 31 gennaio 2014) Il metodo **isMax** accetta un oggetto **x**, un comparatore ed un insieme di oggetti, e restituisce **true** se, in base al comparatore, **x** è maggiore o uguale di tutti gli oggetti contenuti nell'insieme. Altrimenti, il metodo restituisce **false**.

Valutare ciascuna delle seguenti intestazioni per il metodo **isMax**, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie e semplicità. Infine, scegliere l'intestazione migliore oppure proporre un'altra, motivando brevemente la propria scelta.

- (a) **boolean** isMax(Object x, Comparator<Object> c, Set<Object> s)
- (b) <T> **boolean** isMax(T x, Comparator<T> c, Set<T> s)
- (c) <T> **boolean** isMax(T x, Comparator<? **super** T> c, Set<T> s)
- (d) <T> **boolean** isMax(T x, Comparator<? **extends** T> c, Set<? **super** T> s)
- (e) <T> **boolean** isMax(T x, Comparator<? **super** T> c, Set<? **super** T> s)
- (f) <S,T **extends** S> **boolean** isMax(T x, Comparator<? **super** S> c, Set<S> s)

116. (10 punti, 31 gennaio 2014) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Il pattern **Iterator** prevede un metodo che contemporaneamente restituisce il prossimo oggetto e fa avanzare di un posto l'iteratore.
- ☐ ☐ Nel pattern **Decorator** l'oggetto da decorare ha un metodo per aggiungere una decorazione.
- ☐ ☐ Ogni qual volta un metodo accetta come argomento un altro oggetto siamo in presenza del pattern **Strategy**.
- ☐ ☐ Il pattern **Factory Method** prevede che un oggetto ne crei un altro.
- ☐ ☐ Il pattern **Template Method** può essere implementato utilizzando classi astratte.

117. (24 punti, 25 settembre 2013) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, A y) { return "A0"; }
    public String f(A[] x, A y)    { return "A1"; }
    public String f(B[] x, A y)    { return "A2"; }
}
class B extends A {
    public String f(A[] x, A y) { return "B1"; }
    public String f(B x, A y)   { return "B2:" + f((A) x, null); }
    public String f(B[] x, A y) { return "B3"; }
    private String f(A x, Object y) { return "B4"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
```

```

    A alfa = beta;
    B[] arr = new B[10];
    System.out.println(alfa.f(null, alfa));
    System.out.println(beta.f(arr, alfa));
    System.out.println(beta.f(arr, beta));
    System.out.println(234 & 234);
}
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

118. **Movie.** (35 punti, 25 settembre 2013) La classe `Movie` rappresenta un film, con attributi titolo (stringa) e anno di produzione (intero). Alcuni film formano delle serie, cioè sono dei *sequel* di altri film. La classe offre due costruttori: uno per film normali e uno per film appartenenti ad una serie. Quest'ultimo costruttore accetta come terzo argomento il film di cui questo è il successore. Il metodo `getSeries` restituisce la lista dei film che formano la serie a cui questo film appartiene. Se invocato su un film che non appartiene ad una serie, il metodo restituisce una lista contenente solo questo film.

Il metodo statico `selectByYear` restituisce l'insieme dei film prodotti in un dato anno, in tempo costante.

Esempio d'uso:	Output:
<pre> Movie r1 = new Movie("Rocky", 1976); Movie r2 = new Movie("Rocky_II", 1979, r1); Movie r3 = new Movie("Rocky_III", 1982, r2); Movie f = new Movie("Apocalypse_Now", 1979); Set<Movie> movies1979 = Movie.selectByYear(1979); System.out.println(movies1979); List<Movie> rockys = r2.getSeries(); System.out.println(rockys); </pre>	<pre> [Rocky II, Apocalypse Now] [Rocky, Rocky II, Rocky III] </pre>

119. **executeWithDeadline.** (28 punti, 25 settembre 2013) Implementare il metodo statico `executeWithDeadline`, che accetta un riferimento r a un `Runnable` ed un tempo t espresso in secondi. Il metodo esegue r fino ad un tempo massimo di t secondi, trascorsi i quali il metodo interromperà r e restituirà il controllo al chiamante.

Quindi, il metodo deve terminare quando una delle seguenti condizioni diventa vera: 1) il `Runnable` termina la sua esecuzione, oppure 2) trascorrono t secondi.

Si può supporre che il `Runnable` termini quando viene interrotto.

120. **composeMaps.** (28 punti, 25 settembre 2013) Il metodo `composeMaps` accetta due mappe a e b , e restituisce una nuova mappa c così definita: le chiavi di c sono le stesse di a ; il valore associato in c ad una chiave x è pari al valore associato nella mappa b alla chiave $a(x)$.

Nota: Se consideriamo le mappe come funzioni matematiche, la mappa c è definita come $c(x) = b(a(x))$, cioè come composizione di a e b .

Valutare ciascuna delle seguenti intestazioni per il metodo `composeMaps`, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, specificità del tipo di ritorno e semplicità. Infine, scegliere l'intestazione migliore oppure proporre un'altra.

(a) `<S, T, U> Map<S,U> composeMaps(Map<S, T> a, Map<T, U> b)`

- (b) `<S, T, U> Map<S,U> composeMaps(Map<S, T> a, Map<? extends T, U> b)`
- (c) `<S, T, U> Map<S,U> composeMaps(Map<S, T> a, Map<? super T, U> b)`
- (d) `<S, U> Map<S,U> composeMaps(Map<S, ?> a, Map<?, U> b)`
- (e) `<S, U> Map<S,U> composeMaps(Map<S, Object> a, Map<Object, U> b)`

121. (10 punti, 25 settembre 2013) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Il pattern **Composite** consente di aggregare gli oggetti in una struttura ad albero.
- ☐ ☐ In Java, il pattern **Composite** prevede che oggetti primitivi e composti implementino la stessa interfaccia.
- ☐ ☐ Nel pattern **Decorator**, l'oggetto da decorare ha un metodo che aggiunge una decorazione.
- ☐ ☐ Il pattern **Observer** può essere utilizzato per notificare gli eventi generati da un'interfaccia grafica.
- ☐ ☐ Il metodo `hashCode` di `Object` rappresenta un'istanza del pattern **Template Method**.

122. (30 punti, 16 dicembre 2013) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, A y) { return "A1"; }
    public String f(A[] x, A y) { return "A2"; }
    public String f(Object[] x, B y) { return "A3"; }
}
class B extends A {
    public String f(A[] x, A y) { return "B1"; }
    public String f(B[] x, A y) { return "B2:" + f(x, null); }
    public String f(B[] x, B y) { return "B3"; }
}
public class Test {
    public static void main(String[] args) {
        A[] arrA = new B[20];
        B[] arrB = new B[10];
        arrA[0] = arrB[0] = new B();
        System.out.println(arrA[0].f(null, arrB[0]));
        System.out.println(arrB[0].f(arrA, arrB[0]));
        System.out.println(arrB[0].f(arrB, arrA[0]));
        System.out.println(3 | 4);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

123. **Note.** (22 punti, 16 dicembre 2013) Realizzare la classe enumerata **Note**, che rappresenta le sette note musicali. Le note sono disposte su una scala di frequenze, in modo che ciascuna nota sia separata dalla successiva da due *semitoni*, tranne il Mi, che è separato dal Fa da un solo semitono. La classe offre il metodo `interval`, che accetta un altro oggetto `Note x` e restituisce il numero di semitoni tra questa nota e la nota `x`.

Si faccia in modo che il metodo `interval` funzioni in tempo costante, cioè indipendente dall'argomento che gli viene passato.

Esempio d'uso:	Output:
Note a = Note.DO;	4
System.out.println(a.interval(Note.MI));	5
System.out.println(Note.MI.interval(Note.LA));	-2
System.out.println(Note.LA.interval(Note.SOL));	

124. **concurrentMax.** (35 punti, 16 dicembre 2013) Implementare il metodo statico **concurrentMax**, che accetta un riferimento ad una matrice di interi e restituisce il massimo valore presente nella matrice. Internamente, il metodo crea tanti thread quante sono le righe della matrice. Ciascuno di questi thread ricerca il massimo all'interno della sua riga e poi aggiorna il massimo globale.

È necessario evitare *race condition* ed attese attive.

Esempio d'uso:	Output:
<pre>int [][] arr = { {23, 23, 45, 2, 4}, {-10, 323, 33, 445, 4}, {12, 44, 90, 232, 122} };</pre>	445
System.out.println(concurrentMax(arr));	

125. **agree.** (28 punti, 16 dicembre 2013) Il metodo **agree** accetta due oggetti **Comparator** **c1** e **c2** e altri due oggetti **a** e **b**, e restituisce **true** se i due comparatori concordano sull'ordine tra **a** e **b** (cioè, l'esito delle invocazioni **c1(a,b)** e **c2(a,b)** è lo stesso) e **false** altrimenti.

Valutare ciascuna delle seguenti intestazioni per il metodo **agree**, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie e semplicità. Infine, scegliere l'intestazione migliore oppure proporre un'altra, motivando brevemente la propria scelta.

- (a) `<T> boolean agree(Comparator<T> c1, Comparator<T> c2, T a, T b)`
- (b) `boolean agree(Comparator<Object> c1, Comparator<Object> c2, Object a, Object b)`
- (c) `<S, T> boolean agree(Comparator<S> c1, Comparator<T> c2, S a, T b)`
- (d) `<T> boolean agree(Comparator<? extends T> c1, Comparator<? extends T> c2, T a, T b)`
- (e) `<T> boolean agree(Comparator<? super T> c1, Comparator<? super T> c2, T a, T b)`
- (f) `<S, T extends S> boolean agree(Comparator<S> c1, Comparator<S> c2, T a, T b)`

126. (10 punti, 16 dicembre 2013) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ L'annidamento di componenti grafici Swing/AWT sfrutta il pattern **Composite**.
- ☐ ☐ Tipicamente, nel pattern **Decorator** l'oggetto da decorare viene passato al costruttore dell'oggetto decoratore.
- ☐ ☐ Il pattern **Template Method** si applica in presenza di una gerarchia di classi e sotto-classi.
- ☐ ☐ Il pattern **Factory Method** si applica quando un oggetto deve contenerne altri.
- ☐ ☐ Nel pattern **Strategy** la classe **Context** ha un metodo che accetta un oggetto di tipo **Strategy**.

127. (25 punti, 25 giugno 2013) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):


```

class A {
    public String f(Object x, short n) { return "A0"; }
    public String f(A x, int n) { return "A1:" + n; }
    public String f(A x, double n) { return "A2:" + f(x, (int) n); }
    private String f(B x, int n) { return "A3"; }
}
class B extends A {
    public String f(A x, int n) { return "B1:" + n; }
    public String f(B x, double n) { return "B2:" + f((A) x, 2); }
    public String f(A x, float n) { return "B3"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(null, 2L));
        System.out.println(beta.f(beta, 5.0));
        System.out.println(8 | 4);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

128. **String comparator.** (25 punti, 25 giugno 2013) Dire quali delle seguenti sono specifiche valide per un `Comparator` tra due oggetti di tipo `String`, motivando la risposta. Nei casi non previsti dalle specifiche, il comparatore restituisce 0.

`compare(a, b)` restituisce:

- −1 se a contiene caratteri non alfabetici (ad es., numeri) e b no; 1 se b contiene caratteri non alfabetici (ad es., numeri) ed a no.
- −1 se a è lunga esattamente la metà di b ; 1 se a è lunga esattamente il doppio di b .
- −1 se a è un prefisso proprio di b (cioè un prefisso diverso da b); 1 se b è un prefisso proprio di a .
- −1 se b comincia per maiuscola e a no; 1 se sia a sia b cominciano per maiuscola.
- −1 se a contiene la lettera “x” e b contiene la lettera “y”; 1 se a non contiene la lettera “x” e b non contiene la lettera “y”.

129. **MultiBuffer.** (35 punti, 25 giugno 2013) Implementare la classe parametrica `MultiBuffer`, che rappresenta un insieme di buffer. Il suo costruttore accetta il numero n di buffer da creare. Il metodo `insert` inserisce un oggetto nel buffer che in quel momento contiene meno elementi. Il metodo bloccante `take` accetta un intero i compreso tra 0 ed $n - 1$ e restituisce il primo oggetto presente nel buffer i -esimo. La classe deve risultare *thread-safe*.

Esempio d'uso:	Output:
<pre> MultiBuffer<Integer> mb = new MultiBuffer<Integer>(3); mb.insert(13); mb.insert(24); mb.insert(35); System.out.println(mb.take(0)); </pre>	13

Si consideri il seguente schema di sincronizzazione: `insert` è mutuamente esclusivo con `take(i)`, per ogni valore di i ; `take(i)` è mutuamente esclusivo con `take(i)`, ma è compatibile con `take(j)`, quando j è diverso da i . Rispondere alle seguenti domande:

- (a) Questo schema evita le *race condition*?
- (b) E' possibile implementare questo schema utilizzando metodi e blocchi sincronizzati?

130. **Concat.** (30 punti, 25 giugno 2013) Implementare il metodo `concat` che accetta due iteratori `a` e `b` e restituisce un altro iteratore che restituisce prima tutti gli elementi restituiti da `a` e poi tutti quelli di `b`.

Valutare le seguenti intestazioni per il metodo `concat`, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporre un'altra.

- (a) `Iterator<Object> concat(Iterator<Object> a, Iterator<Object> b)`
- (b) `Iterator<?> concat(Iterator<?> a, Iterator<?> b)`
- (c) `<S> Iterator<S> concat(Iterator<S> a, Iterator<S> b)`
- (d) `<S> Iterator<S> concat(Iterator<? extends S> a, Iterator<? extends S> b)`
- (e) `<S,T> Iterator<S> concat(Iterator<S> a, Iterator<T> b)`
- (f) `<S,T extends S> Iterator<T> concat(Iterator<T> a, Iterator<S> b)`

131. (10 punti, 25 giugno 2013) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Il pattern `Iterator` prevede un metodo per far ripartire l'iteratore daccapo.
- ☐ ☐ Il pattern `Observer` evita che gli osservatori debbano controllare periodicamente lo stato dell'oggetto osservato (*polling*).
- ☐ ☐ Di norma, il pattern `Decorator` si applica solo quando l'insieme delle decorazioni possibili è illimitato.
- ☐ ☐ Il pattern `Composite` consente anche agli oggetti primitivi di contenerne altri.
- ☐ ☐ In Java, il pattern `Template Method` viene comunemente implementato usando una classe astratta.

132. (25 punti, 5 marzo 2014) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, A y) { return "A1"; }
    public String f(A x, A y)      { return "A2"; }
    public String f(Object x, B y) { return "A3"; }
}
class B extends A {
    public String f(A x, A y) { return "B1"; }
    public String f(B x, A y) { return "B2"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(null, alfa));
        System.out.println(beta.f(beta, beta));
        System.out.println(beta.f(alfa.f(alfa, alfa), beta));
        System.out.println(5 & 7);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

133. **PeriodicTask.** (30 punti, 5 marzo 2014) Realizzare la classe `PeriodicTask`, che consente di eseguire un `Runnable` periodicamente, ad intervalli specificati. Il costruttore accetta un oggetto `Runnable r` e un numero di millisecondi p , detto *periodo*, e fa partire un thread che esegue `r.run()` ogni p millisecondi (si noti che il costruttore non è bloccante). Il metodo `getTotalTime` restituisce il numero complessivo di millisecondi che tutte le chiamate a `r.run()` hanno utilizzato fino a quel momento.

Suggerimento: il seguente metodo della classe `System` restituisce il numero di millisecondi trascorsi dal primo gennaio 1970: `public static long currentTimeMillis()`.

(15 punti) Inoltre, dire quali dei seguenti criteri di uguaglianza per oggetti di tipo `PeriodicTask` sono validi, giustificando brevemente la risposta. Due oggetti di tipo `PeriodicTask` sono uguali se:

- hanno lo stesso `Runnable` ed un periodo inferiore ad un secondo;
- hanno due periodi che sono l'uno un multiplo intero dell'altro (ad es. 5000 millisecondi e 2500 millisecondi);
- hanno lo stesso `Runnable` oppure lo stesso periodo.

Esempio d'uso:	Output:
<pre>Runnable r = new Runnable() { public void run() { System.out.println("Ciao!"); } }; new PeriodicTask(r, 2000);</pre>	<pre>Ciao! Ciao! (dopo 2 secondi) Ciao! (dopo altri 2 secondi) ...</pre>

134. **Status.** (17 punti, 5 marzo 2014) Implementare la classe enumerata `Status`, che rappresenta le 4 modalità di un programma di *instant messaging*: `ONLINE`, `BUSY`, `HIDDEN` e `OFFLINE`. Il metodo `isVisible` restituisce vero per i primi due e falso per gli altri. Il metodo `canContact` accetta un altro oggetto `Status x` e restituisce vero se un utente in questo stato può contattare un utente nello stato x e cioè se questo stato è diverso da `OFFLINE` e lo stato x è visibile.

Esempio d'uso:	Output:
<pre>Status a = Status.BUSY, b = Status.HIDDEN; System.out.println(a.isVisible()); System.out.println(a.canContact(b));</pre>	<pre>true false</pre>

135. **isMax.** (28 punti, 5 marzo 2014) Il metodo `extractPos` accetta una lista ed un numero intero n e restituisce l'ennesimo elemento della lista.

Valutare ciascuna delle seguenti intestazioni per il metodo `extractPos`, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra, motivando brevemente la propria scelta.

- `Object extractPos(Collection<?> l, int n)`
- `<T> T extractPos(List<T> l, int n)`
- `<T> T extractPos(List<? extends T> l, int n)`
- `Object extractPos(List<?> l, int n)`
- `<T> T extractPos(LinkedList<T> l, int n)`

(f) `<S,T> S extractPos(List<T> l, int n)`

136. (10 punti, 5 marzo 2014) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Il pattern `Iterator` consente ad una collezione di scorrere i propri elementi senza esporre la sua struttura interna.
- ☐ ☐ Nel pattern `Observer` l'oggetto osservato conserva dei riferimenti ai suoi osservatori.
- ☐ ☐ Il pattern `Strategy` usa un oggetto per rappresentare una variante di un algoritmo.
- ☐ ☐ Passare un `Comparator` al metodo `Collections.sort` rappresenta un'istanza del pattern `Strategy`.
- ☐ ☐ Il metodo `hashCode` di `Object` rappresenta un'istanza del pattern `Template Method`.

137. (25 punti, 29 aprile 2013) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, Object y) { return "A1"; }
    public String f(Object x, B y) { return "A2"; }
}
class B extends A {
    private String f(B x, A y) { return "B1"; }
    public String f(Object x, B y) { return "B2"; }
}
class C extends B {
    public String f(B x, A y) { return "C1"; }
    public String f(A x, Object y) { return "C2:" + f(null, x); }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(beta.f(null, beta));
        System.out.println(alfa.f(gamma, alfa));
        System.out.println(gamma.f(beta, alfa));
        System.out.println(5 | 8);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

138. **City.** (30 punti, 29 aprile 2013) La classe `City` rappresenta una città. Il costruttore accetta il nome della città, mentre il metodo `connect` accetta un'altra città e stabilisce un collegamento tra le due (una strada o un altro tipo di collegamento). Tutti i collegamenti sono bidirezionali.

Il metodo `getConnections` restituisce la collezione delle città *direttamente* collegate a questa. Il metodo `isConnected` prende come argomento un'altra città e restituisce vero se è collegata a `this` *direttamente* o *indirettamente* (cioè, tramite un numero arbitrario di collegamenti).

Esempio d'uso:	Output:
<pre>City n = new City("Napoli"), r = new City("Roma"), s = new City("Salerno"), p = new City("Parigi"); n.connect(s); n.connect(r); Collection<City> r_conn = r.getConnections(); System.out.println(r_conn); System.out.println(r.isConnected(s)); System.out.println(r.isConnected(p));</pre>	<pre>[Napoli] true false</pre>

139. **Pair.** (25 punti, 29 aprile 2013) Realizzare la classe parametrica **Pair**, che rappresenta una coppia di oggetti di tipo potenzialmente diverso. La classe deve supportare le seguenti funzionalità:

- 1) due **Pair** sono uguali secondo **equals** se entrambe le loro componenti sono uguali secondo **equals**;
- 2) il codice hash di un oggetto **Pair** è uguale allo XOR tra i codici hash delle sue due componenti;
- 3) la stringa corrispondente ad un oggetto **Pair** è **"(str1,str2)"**, dove **str1** (rispettivamente, **str2**) è la stringa corrispondente alla prima (risp., seconda) componente.

Esempio d'uso:	Output:
<pre>Pair<String,Integer> p1 = new Pair<String,Integer>("uno", 1); System.out.println(p1);</pre>	<pre>(uno,1)</pre>

140. (15 punti, 29 aprile 2013) Con riferimento alla classe **Pair** dell'esercizio 2, dire quali delle seguenti specifiche per il metodo **equals** sono valide e perché.

Due istanze *a* e *b* di **Pair** sono uguali se...

- (a) ...hanno almeno una delle due componenti uguale.
- (b) ...hanno una delle due componenti uguale e una diversa.
- (c) ...la prima componente di *a* è uguale alla seconda componente di *b* e la seconda componente di *a* è uguale alla prima componente di *b*.

141. (30 punti, 9 luglio 2013) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, double n) { return "A0"; }
    public String f(A[] x, int n)      { return "A1"; }
    public String f(B[] x, float n)    { return "A2:" + f(x[0], (int) n); }
}
class B extends A {
    public String f(A[] x, int n)      { return "B1:" + n; }
    public String f(B x, double n)    { return "B2:" + f((A) x, 2); }
    public String f(B[] x, float n)   { return "B3"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        B[] arr = new B[10];
        System.out.println(alfa.f(null, 2L));
        System.out.println(beta.f(arr, 5.0));
        System.out.println(beta.f(arr, 2));
        System.out.println(11 ^ 11);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

142. **BloodType.** (20 punti, 9 luglio 2013) Implementare l'enumerazione `BloodType`, che rappresenta i quattro gruppi sanguigni del sistema AB0, e cioè: A, B, AB e 0.

Il metodo `canReceiveFrom` accetta un altro oggetto `BloodType` x e restituisce `true` se questo gruppo sanguigno può ricevere trasfusioni dal gruppo x e `false` altrimenti.

Si ricordi che ogni gruppo può ricevere dal gruppo stesso e dal gruppo 0. In più, il gruppo AB può ricevere anche da A e da B (quindi, da tutti).

143. **processArray.** (40 punti, 9 luglio 2013) L'interfaccia `RunnableFunction` rappresenta una funzione che accetta un parametro e restituisce un valore dello stesso tipo.

```
interface RunnableFunction<T> {
    public T run(T x);
}
```

Implementare il metodo statico `processArray`, che esegue una data `RunnableFunction` f su tutti gli elementi di un array di input e memorizza i risultati in un altro array, di output. Inoltre, il metodo riceve come argomento un intero n ed utilizza n thread diversi per eseguire la funzione f contemporaneamente su diversi elementi dell'array.

Ad esempio, se $n = 2$, il metodo potrebbe lanciare due thread che eseguono f sui primi due elementi dell'array. Poi, appena uno dei due thread termina, il metodo potrebbe lanciare un nuovo thread che esegue f sul terzo elemento dell'array, e così via.

Rispondere alla seguente domanda: nella vostra implementazione, quand'è che il metodo `processArray` restituisce il controllo al chiamante?

Esempio d'uso:	Output:
<pre>String[] x = {"uno", "due", "tre"}, y = new String[3]; RunnableFunction<String> f = new RunnableFunction<String>() { public String run(String x) { return x + x; } }; processArray(x, y, f, 2); for (String s: y) System.out.println(s);</pre>	<pre>ununo duedue treetre</pre>

144. **isSorted.** (25 punti, 9 luglio 2013) Implementare il metodo `isSorted` che accetta una collezione e un comparatore, e restituisce `true` se la collezione risulta già ordinata in senso non-decrescente rispetto a quel comparatore, e `false` altrimenti.

Valutare ciascuna delle seguenti intestazioni per il metodo `isSorted`, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie e semplicità. Infine, scegliere l'intestazione migliore oppure proporre un'altra.

- `boolean isSorted(Collection<?> x, Comparator<Object> c)`
- `<S> boolean isSorted(Collection<? extends S> x, Comparator<? super S> c)`
- `<S> boolean isSorted(Collection<S> x, Comparator<S> c)`
- `boolean isSorted(Collection<Object> x, Comparator<Object> c)`
- `<S, T> boolean isSorted(Collection<S> x, Comparator<T> c)`
- `<S, T extends S> boolean isSorted(Collection<T> x, Comparator<S> c)`

145. (10 punti, 9 luglio 2013) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Il pattern **Composite** prevede che si possa iterare sui componenti di un oggetto composto.
- ☐ ☐ Nel pattern **Decorator**, l'oggetto da decorare ha un metodo che aggiunge una decorazione.
- ☐ ☐ Nel pattern **Observer**, il soggetto osservato mantiene riferimenti a tutti gli osservatori.
- ☐ ☐ Nel pattern **Observer**, gli osservatori hanno un metodo per registrare un soggetto da osservare.
- ☐ ☐ La scelta del layout di un container **AWT** rappresenta un'istanza del pattern **Strategy**.
146. **Color.** (33 punti, 22 gennaio 2010) La classe **Color** rappresenta un colore, determinato dalle sue componenti RGB. La classe offre alcuni colori predefiniti, tra cui **RED**, **GREEN** e **BLUE**. Un colore nuovo si può creare solo con il metodo factory **make**. Se il client cerca di ricreare un colore predefinito, gli viene restituito quello e non uno nuovo. Ridefinire anche il metodo **toString**, in modo che rispetti il seguente caso d'uso.

Esempio d'uso:	Output dell'esempio d'uso:
<pre>Color rosso = Color.RED; Color giallo = Color.make(255, 255, 0); Color verde = Color.make(0, 255, 0); System.out.println(rosso); System.out.println(giallo); System.out.println(verde); System.out.println(verde == Color. GREEN);</pre>	<pre>red (255, 255, 0) green true</pre>

147. (27 punti, 22 gennaio 2010) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A { public String f(int[] a, int l)    { return "A1"; }
        public String f(int[] a, double d) { return "A2"; }
        public String f(Object o, int l)   { return "A3"; }
}
class B extends A {
    public String f(double[] a, double d) { return "B1"; }
}
class C extends B {
    public final String f(int[] a, int l) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta  = new B();
        A alfa  = gamma;
        int[] x = new int[10];
        System.out.println(alfa.f(x, 10));
        System.out.println(beta.f(x, x[1]));
        System.out.println(gamma.f(null, 10));
        System.out.println(gamma.f(x, 3.0));
        System.out.println(alfa instanceof C);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (12 punti)

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (15 punti)
148. **GetByType.** (15 punti, 22 gennaio 2010) Implementare il metodo statico `getByType` che, data una collezione `c` (`Collection`) ed un oggetto `x` di tipo `Class`, restituisce un oggetto della collezione il cui tipo effettivo sia esattamente `x`. Se un tale oggetto non esiste, il metodo restituisce `null`.
Prestare particolare attenzione alla scelta della firma del metodo. Si ricordi che la classe `Class` è parametrica.
149. (15 punti, 22 gennaio 2010) La seguente classe `A` fa riferimento ad una classe `B`. Implementare la classe `B` in modo che venga compilata correttamente e permetta la compilazione della classe `A`.

```
public class A extends B {

    public A() {
        b1 = new B.C(true);
        b2 = new B(false);
    }

    public B f(Object o) {
        B x = super.f(o);
        return x.clone();
    }

    private B.C c = new B.C(3);
    private B b1, b2;
}
```

150. (20 punti, 22 gennaio 2010) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ La classe `Object` ha un metodo `sleep`.
 - ☐ ☐ Il metodo `wait` può lanciare un'eccezione verificata.
 - ☐ ☐ Il metodo `notify` sveglia tutti i thread in attesa su quest'oggetto.
 - ☐ ☐ Un metodo statico non può essere `synchronized`.
 - ☐ ☐ La classe `HashMap<K,V>` estende l'interfaccia `Collection<K>`.
 - ☐ ☐ Una classe enumerata (`enum`) può estenderne un'altra.
 - ☐ ☐ Nel pattern `Observer`, ogni osservatore conosce tutti gli altri.
 - ☐ ☐ La scelta del layout di una finestra in AWT rappresenta un'applicazione del pattern `Strategy`.
 - ☐ ☐ Il pattern `Composite` permette di trattare un insieme di elementi come un elemento primitivo.
 - ☐ ☐ Il pattern `Decorator` si applica quando l'insieme delle decorazioni possibili è illimitato.
151. **Auction.** (33 punti, l'8 settembre 2009) La classe `Auction` rappresenta una vendita all'asta. Il suo costruttore accetta come argomento il prezzo di partenza dell'asta. Il metodo `makeOffer` rappresenta la presentazione di un'offerta e prende come argomenti l'ammontare dell'offerta e il nome dell'acquirente.

Un oggetto `Auction` deve accettare offerte, finchè non riceve offerte per 3 secondi consecutivi. A quel punto, l'oggetto stampa a video l'offerta più alta e il nome del compratore.

Si supponga che più thread possano chiamare concorrentemente il metodo `makeOffer` dello stesso oggetto.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> Auction a = new Auction(1000) ; a.makeOffer(1100, 'Marco'); a.makeOffer(1200, 'Luca'); Thread.sleep(1000); a.makeOffer(200, 'Anna'); Thread.sleep(1000); a.makeOffer(1500, 'Giulia'); ; Thread.sleep(4000); </pre>	<p>Oggetto venduto a Giulia per 1500 euro.</p>

152. (27 punti, l'8 settembre 2009) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A { public String f(double n, A x) { return "A1"; }
        public String f(double n, B x) { return "A2"; }
        public String f(int n, Object x) { return "A3"; }
}
class B extends A {
    public String f(double n, B x) { return "B1"; }
    public String f(float n, Object y) { return "B2"; }
}
class C extends A {
    public final String f(int n, Object x) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(3, beta));
        System.out.println(alfa.f(3.0, beta));
        System.out.println(beta.f(3.0, alfa));
        System.out.println(gamma.f(3, gamma));
        System.out.println(false || alfa.equals(beta));
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (12 punti)
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (15 punti)
153. **IncreasingSubsequence.** (25 punti, l'8 settembre 2009) Implementare la classe `IncreasingSubseq` che, data una lista di oggetti tra loro confrontabili, rappresenta la *sottosequenza crescente* che inizia col primo elemento.

Attenzione: la classe deve funzionare con qualunque tipo di dato che sia confrontabile (non solo con "Integer").

Sarà valutato negativamente l'uso di "strutture di appoggio", ovvero di spazio aggiuntivo di dimensione non costante.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> List<Integer> l = new LinkedList<Integer>(); l.add(10); l.add(3); l.add(5); l.add(12); l.add(11); l.add(35); for (Integer i: new IncreasingSubseq<Integer>(l)) System.out.println(i); </pre>	<p>10 12 35</p>

154. (15 punti, 1°8 settembre 2009) La seguente classe **A** fa riferimento ad una classe **B**. Implementare la classe **B** in modo che venga compilata correttamente e permetta la compilazione della classe **A**.

```
public class A {
    private A a = new B.C(3);
    private double x = a.f(3);
    private B b = new B.D(3);

    private int f(int n) {
        g(new B(3), n);
        return 2*n;
    }
    private void g(A u, int z) { }
    private void g(B u, double z) { }

    public A(int i) { }
}
```

155. (20 punti, 1°8 settembre 2009) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ `equals` è un metodo `final` della classe `Object`.
- ☐ ☐ Una classe può implementare più interfacce contemporaneamente.
- ☐ ☐ Il metodo `add` di `Set` funziona sempre in tempo costante.
- ☐ ☐ Il metodo `add` di `List` non restituisce mai `false`.
- ☐ ☐ Il modificatore `transient` indica che quel campo non deve essere serializzato.
- ☐ ☐ Le classi enumerate non possono essere istanziate con `new`.
- ☐ ☐ Il pattern `Decorator` permettere di aggiungere funzionalità ad una classe.
- ☐ ☐ Inserire una serie di oggetti in una `LinkedList` rappresenta un'istanza del pattern `Composite`.
- ☐ ☐ Il pattern `Factory Method` suggerisce che il "prodotto generico" sia sottotipo del "produttore generico".
- ☐ ☐ Ridefinire il metodo `clone` tramite overriding rappresenta un'istanza del pattern `Strategy`.

156. **Triangle 2.** (33 punti, 27 novembre 2009) La classe `Triangle` rappresenta un triangolo. Il suo costruttore accetta la misura dei suoi lati, e lancia un'eccezione se tali misure non danno luogo ad un triangolo. Il metodo `equals` stabilisce se due triangoli sono isometrici (uguali). Il metodo `similar` stabilisce se due triangoli sono simili (hanno gli stessi angoli, ovvero lo stesso rapporto tra i lati).

Il metodo `perimeterComparator` restituisce un comparatore che confronta i triangoli in base al loro perimetro.

Nota: tre numeri positivi x , y e z possono essere le misure dei lati di un triangolo a patto che $x < y + z$, $y < x + z$ e $z < x + y$.

Esempio d'uso:	Output dell'esempio d'uso:
<pre>Triangle a = new Triangle(3,4,5); Triangle b = new Triangle(4,5,3); Triangle c = new Triangle(8,6,10); System.out.println(a.equals(b)); System.out.println(a.similar(b)); System.out.println(a.similar(c)); Comparator<Triangle> pc = Triangle. perimeterComparator(); System.out.println(pc.compare(b, c));</pre>	<pre>true true true -1</pre>

157. (27 punti, 27 novembre 2009) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A { public String f(double n, Object x) { return "A1"; }
        public String f(double n, A x)      { return "A2"; }
        public String f(int n,      Object x) { return "A3"; }
}
class B extends A {
    public String f(double n, Object x) { return "B1"; }
    public String f(float n,  Object y) { return "B2"; }
}
class C extends B {
    public final String f(double n, A x) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta  = new B();
        A alfa  = gamma;
        System.out.println(alfa.f(3.0, gamma));
        System.out.println(beta.f(3, beta));
        System.out.println(beta.f(3.0, null));
        System.out.println(gamma.f(3.0, gamma));
        System.out.println(true && (alfa==beta));
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (12 punti)
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (15 punti)
158. **CountByType.** (25 punti, 27 novembre 2009) Implementare il metodo statico `countByType` che, data una lista di oggetti, stampa a video il numero di oggetti contenuti nella lista, *divisi in base al loro tipo effettivo*.

Attenzione: il metodo deve funzionare con qualunque tipo di lista e di oggetti contenuti.

Esempio d'uso:	Output dell'esempio d'uso:
<pre>List<Number> l = new LinkedList<Number>(); l.add(new Integer(3)); l.add(new Double(4.0)) l.add(new Float(7.0f)); l.add(new Integer(11)); countByType(l);</pre>	<pre>java.lang.Double : 1 java.lang.Float : 1 java.lang.Integer : 2</pre>

159. (15 punti, 27 novembre 2009) La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A {
    public static final A a = new B(null);
    public final int n = B.f(3);

    public Object g() {
        B b = new B();
        B.C c = b.new C(7);
        return c;
    }

    public A(int i) { }
}
```

160. (20 punti, 27 novembre 2009) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ La classe `Thread` ha un costruttore che accetta un `Runnable`.
- ☐ ☐ `Runnable` è un'interfaccia vuota.
- ☐ ☐ La serializzazione è un modo standard di memorizzare oggetti su file.
- ☐ ☐ Se due oggetti sono uguali secondo il metodo `equals`, dovrebbero avere lo stesso codice hash secondo il metodo `hashCode`.
- ☐ ☐ L'interfaccia `Comparator<T>` estende l'interfaccia `Comparable<T>`.
- ☐ ☐ Le classi enumerate non possono essere istanziate con `new`.
- ☐ ☐ Nel pattern `Observer`, più oggetti possono osservare lo stesso oggetto.
- ☐ ☐ Il pattern `Composite` prevede un'interfaccia che rappresenti un oggetto primitivo.
- ☐ ☐ Le interfacce `Iterator` e `Iterable` rappresentano un'istanza del pattern `Factory Method`.
- ☐ ☐ Ridefinire il metodo `clone` tramite overriding rappresenta un'istanza del pattern `Template Method`.

161. **Tutor.** (33 punti, 19 giugno 2009) Un *tutor* è un dispositivo per la misurazione della velocità media in autostrada. Una serie di sensori identifica i veicoli in base alle targhe e ne calcola la velocità, misurando il tempo che il veicolo impiega a passare da un sensore al successivo (e, naturalmente, conoscendo la distanza tra i sensori).

Si implementi la classe `Tutor` e la classe `Detector` (sensore). Il metodo `addDetector` di `Tutor` crea un nuovo sensore posto ad un dato kilometro del tracciato. Il metodo `carPasses` di `Detector` rappresenta il passaggio di un veicolo davanti a questo sensore: esso prende come argomenti la targa di un veicolo ed un tempo assoluto in secondi, e restituisce una stima della velocità di quel veicolo, basata anche sui dati dei sensori che lo precedono. Tale metodo restituisce `-1` se il sensore non ha sufficienti informazioni per stabilire la velocità.

Si supponga che le chiamate ad `addDetector` avvengano tutte all'inizio e con kilometri crescenti, come nel seguente esempio d'uso.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> Tutor tang = new Tutor(); Tutor.Detector a = tang.addDetector(2); Tutor.Detector b = tang.addDetector(5); Tutor.Detector c = tang.addDetector(9); // nuovo veicolo System.out.println(a.carPasses("NA12345", 0)); // 3km in 1200 sec (20 minuti), quindi 9km/h System.out.println(b.carPasses("NA12345", 1200)); // nuovo veicolo System.out.println(b.carPasses("SA00001", 1200)); // 4km in 120 sec (2 minuti), quindi 120km/h System.out.println(c.carPasses("NA12345", 1320)); // 4km in 180 sec (3 minuti), quindi 80km/h System.out.println(c.carPasses("SA00001", 1380)); </pre>	<pre> -1 9 -1 120 80 </pre>

162. (27 punti, 19 giugno 2009) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    String f(A x, A y) { return "A1"; }
    String f(B x, B y) { return "A2"; }
}
class B extends A {
    String f(B x, B y) { return "B1:" + x.f(x,y); }
}
class C extends B {
    String f(B x, B y) { return "C1"; }
    String f(B x, Object y) { return "C2"; }
    String f(C x, Object y) { return "C3"; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;
        System.out.println( alfa.f( alfa , beta ));
        System.out.println( beta.f(gamma, beta ));
        System.out.println( gamma.f(gamma, alfa ));
        System.out.println( gamma.f( alfa , gamma ));
        int x=0;
        System.out.println( (true || (x++>0)) + ":" + x );
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (15 punti)
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)
163. **Cardinal.** (22 punti, 19 giugno 2009) Implementare l'enumerazione `Cardinal`, che rappresenta le 16 direzioni della rosa dei venti. Il metodo `isOpposite` prende come argomento un punto cardinale x e restituisce vero se questo punto cardinale è diametralmente opposto ad x , e falso altrimenti. Il metodo statico `mix` prende come argomento due punti cardinali, non opposti, e restituisce il punto cardinale intermedio tra i due.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> Cardinal nord = Cardinal.N; System.out.println(nord.isOpposite(Cardinal.S)); Cardinal nordest = Cardinal.mix(Cardinal.N, Cardinal.E); assert nordest==Cardinal.NE : "Errore_inaspettato!"; Cardinal nordnordest = Cardinal.mix(nordest, Cardinal.N); System.out.println(nordnordest); </pre>	<pre> true NNE </pre>

164. (18 punti, 19 giugno 2009) La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A {
    private List<? extends String> l = B.getList();

    public <T> void f(T x, Comparator<? super T> y) {
        y.compare(x, B.getIt(x));
    }
    public void g(Set<? super Integer> s) {
        Set<String> s2 = B.convert(s);
        f(new B(), B.something);
        f(new Integer(4), B.something);
    }
}
```

165. (20 punti, 19 giugno 2009) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ `sleep` è un metodo statico della classe `Object`.
- ☐ ☐ Il metodo `wait` di `Object` prende un argomento.
- ☐ ☐ Il metodo `clone` di `Object` effettua una copia superficiale.
- ☐ ☐ L'interfaccia `Serializable` è vuota.
- ☐ ☐ Un'eccezione verificata non può essere catturata.
- ☐ ☐ `List<Integer>` è sottotipo di `Iterable<Integer>`.
- ☐ ☐ Il pattern `Template Method` si può applicare quando un algoritmo utilizza delle *operazioni primitive*.
- ☐ ☐ Nel framework MVC, le classi controller gestiscono gli eventi dovuti all'interazione con l'utente.
- ☐ ☐ Il metodo `iterator` dell'interfaccia `Iterable` rappresenta un'istanza del pattern `Factory Method`.
- ☐ ☐ L'aggiunta di un `Component AWT` dentro un altro `Component` rappresenta un'istanza del pattern `Decorator`.

166. **Wall.** (36 punti, 24 febbraio 2010) La classe `Wall` rappresenta un muro di mattoni, ciascuno lungo 10cm, poggiati l'uno sull'altro. Il costruttore accetta l'altezza massima (in file di mattoni) e la larghezza massima (in cm) del muro. Il metodo `addBrick` aggiunge un mattone alla fila e alla posizione (in cm) specificata, restituendo un oggetto di tipo `Brick`. Il metodo `isStable` della classe `Brick` restituisce vero se in quel momento questo mattone è in una posizione stabile, indipendentemente dai mattoni eventualmente poggiati *sopra* di esso.

Esempio d'uso:	Output dell'esempio d'uso: (Nota: l'esempio è accompagnato da una figura alla lavagna)
<pre>Wall w = new Wall(10, 100); w.addBrick(0,10); w.addBrick(0,30); Wall.Brick b3 = w.addBrick(1,2); Wall.Brick b4 = w.addBrick(1,13); Wall.Brick b5 = w.addBrick(1,36); System.out.println(b3.isStable()); System.out.println(b4.isStable()); System.out.println(b5.isStable()); w.addBrick(0,45); System.out.println(b5.isStable());</pre>	<pre>false true false true</pre>

167. (27 punti, 24 febbraio 2010) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A { public String f(int a, int b, float c)    { return "A1"; }
          public String f(int a, double b, float c) { return "A2"; }
}
class B extends A {
    public String f(int a, int b, float c) { return "B1"; }
    private String f(double a, float b, int c) { return "B2"; }
    public String f(double a, int b, float c) { return "B3"; }
}
class C extends B {
    public String f(int a, int b, float c) { return "C1"; }
    public String f(double a, float b, int c) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta  = new B();
        A alfa  = gamma;
        System.out.println(alfa.f(1, 2, 3));
        System.out.println(beta.f(1.0, 2, 3));
        System.out.println(gamma.f(1, 2, 3));
        System.out.println(gamma.f(1.0, 2, 3));
        System.out.println(beta instanceof A);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (12 punti)
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (15 punti)
168. **Version.** (27 punti, 24 febbraio 2010) La classe `Version` rappresenta una versione di un programma. Una versione può avere due o tre parti intere ed, opzionalmente, un'etichetta "alpha" o "beta". (15 punti)

La classe `Version` deve implementare l'interfaccia `Comparable<Version>`, in modo che una versione sia minore di un'altra se la sua numerazione è precedente a quella dell'altra. Le etichette "alpha" e "beta" non influiscono sull'ordinamento. (12 punti)

Rispettare il seguente caso d'uso, compreso il formato dell'output.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> Version v1 = new Version(1, 0); Version v2 = new Version(2, 4, Version.alpha); Version v3 = new Version(2, 6, 33); Version v4 = new Version(2, 6, 34, Version.beta); System.out.println(v1); System.out.println(v2); System.out.println(v3); System.out.println(v4); System.out.println(v1.compareTo(v2)); System.out.println(v4.compareTo(v3)); </pre>	<pre> 1.0 2.4alpha 2.6.33 2.6.34beta -1 1 </pre>

169. (20 punti, 24 febbraio 2010) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ E' possibile sincronizzare un blocco di codice rispetto a qualsiasi oggetto.
- ☐ ☐ Il metodo `wait` mette in attesa il thread corrente.
- ☐ ☐ Una classe enumerata (`enum`) può implementare un'interfaccia.
- ☐ ☐ Una classe astratta può estendere un'altra classe astratta.
- ☐ ☐ Di un metodo `final` non è possibile fare l'overloading.
- ☐ ☐ Un costruttore non può lanciare eccezioni.
- ☐ ☐ Nel pattern `Observer`, il soggetto osservato avvisa gli osservatori degli eventi rilevanti.
- ☐ ☐ Nel framework MVC, le classi `view` si occupano di presentare i dati all'utente.
- ☐ ☐ Il pattern `Template Method` si applica quando un algoritmo si basa su determinate operazioni primitive.
- ☐ ☐ Il pattern `Factory Method` si applica quando una classe deve costruire oggetti di un'altra classe.

170. **UML.** (30 punti, 23 aprile 2009) Nell'ambito di un programma per la progettazione del software, si implementino la classi `UMLClass` e `UMLAggregation`, che rappresentano una classe ed una relazione di aggregazione, all'interno di un diagramma delle classi UML. Il costruttore di `UMLAggregation` accetta le due classi tra le quali vale l'aggregazione, la cardinalità minima e quella massima.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> UMLClass impianto = new UMLClass('Impianto'); UMLClass apparecchio = new UMLClass('Apparecchio'); UMLClass contatore = new UMLClass('Contatore'); new UMLAggregation(apparecchio, impianto, 1, 1); new UMLAggregation(impianto, apparecchio, 0, UMLAggregation.INFINITY); new UMLAggregation(impianto, contatore, 0, 1); System.out.println(impianto); </pre>	<pre> Classe: Impianto Aggregazioni: Impianto-Apparecchio, cardinalita': 0..infinito Impianto-Contatore, cardinalita': 0..1 </pre>

171. (25 punti, 23 aprile 2009) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    String f(A other, int n) { return "A1:" + n; }
    String f(B other, long n) { return "A2:" + n; }
}
class B extends A {
    String f(A other, int n) { return "B1:" + n; }
}
class C extends B {
    String f(B other, long n) { return "C1:" + n; }
    String f(B other, double n) { return "C2:" + n; }
    private String f(C other, long n) { return "C3:" + n; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
    }
}

```



```

    B beta  = gamma;
    A alfa  = gamma;
    System.out.println( alfa.f( alfa , 4L));
    System.out.println( beta.f(gamma, 4L));
    System.out.println( gamma.f( null, 3L));
    System.out.println( 7 >> 1);
}
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (16 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (9 punti)

172. **Circle.** (30 punti, 23 aprile 2009) Nell'ambito di un programma di geometria, la classe `Circle` rappresenta una circonferenza sul piano cartesiano. Il suo costruttore accetta le coordinate del centro ed il valore del raggio. Il metodo `overlaps` prende come argomento un'altra circonferenza e restituisce vero se e solo se le due circonferenze hanno almeno un punto in comune.

Fare in modo che `Circle` implementi `Comparable`, con il seguente criterio di ordinamento: una circonferenza è "minore" di un'altra se è interamente contenuta in essa, mentre se nessuna delle due circonferenze è contenuta nell'altra, esse sono considerate "uguali". Dire se tale criterio di ordinamento è valido, giustificando la risposta.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> Circle c1 = new Circle(0,0,2); Circle c2 = new Circle(1,1,1); System.out.println(c1.overlaps(c2)); System.out.println(c1.compareTo(c2)); </pre>	<pre> true 0 </pre>

173. (15 punti, 23 aprile 2009) La seguente classe `A` fa riferimento ad una classe `B`. Implementare la classe `B` in modo che venga compilata correttamente e permetta la compilazione della classe `A`.

```

public class A {
    private B b1 = new B(null);
    private B b2 = new B.C();
    private B b3 = b1.new D();

    private int f(Object x) {
        if (x==null) throw b2;

        long l = b1.g();
        return b1.g();
    }
}

```

174. **Washer.** (33 punti, 9 luglio 2009) La seguente classe rappresenta le operazioni elementari di una lavatrice:

```

class Washer {
    public void setTemp(int temp) { System.out.println("Setting_temperature
        _to_" + temp); }
    public void setSpeed(int rpm) { System.out.println("Setting_speed_to_"
        + rpm); }
    public void addSoap() { System.out.println("Adding_soap!"); }
}

```

Si implementi una classe `Program`, che rappresenta un programma di lavaggio per una lavatrice. Il metodo `addAction` aggiunge una nuova operazione elementare al programma. Un'operazione elementare può essere una delle tre operazioni elementari della lavatrice, oppure l'operazione "Wait", che aspetta un dato numero di minuti. Il metodo `execute` applica il programma ad una data lavatrice.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> Washer w = new Washer(); Program p = new Program(); p.addAction(new Program. SetTemp(30)); p.addAction(new Program. SetSpeed(20)); p.addAction(new Program.Wait (10)); p.addAction(new Program. AddSoap()); p.addAction(new Program. SetSpeed(100)); p.addAction(new Program.Wait (10)); p.addAction(new Program. SetSpeed(0)); p.execute(w); </pre>	<pre> Setting temperature to 30 Setting speed to 20 Adding soap! (dopo 10 minuti) Setting speed to 100 Setting speed to 0 (dopo 10 minuti) </pre>

175. (27 punti, 9 luglio 2009) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A { public String f(double n, A x, A y) { return "A1"; }
        public String f(double n, B x, B y) { return "A2"; }
        public String f(int n, B x, B y) { return "A3"; }
}
class B extends A {
    public String f(int n, B x, B y) { return "B1:" + x.f(3.0,x,y); }
    public String f(float n, A x, Object y) { return "B2"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(3, alfa, beta));
        System.out.println(alfa.f(4, beta, beta));
        System.out.println(beta.f(3, alfa, (Object) alfa));
        System.out.println(true && (alfa instanceof B));
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (15 punti)
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)
176. **Elevator.** (27 punti, 9 luglio 2009) Implementare la classe `Elevator`, che simula il comportamento di un ascensore. Il costruttore prende come argomento il numero di piani serviti (oltre al pian terreno). Il metodo `call` rappresenta la prenotazione ("chiamata") di un piano. Se l'argomento di `call` è fuori dall'intervallo corretto, viene lanciata un'eccezione.

In un thread indipendente, quando ci sono chiamate in attesa, l'ascensore cambia piano in modo da soddisfare una delle chiamate, scelta in ordine arbitrario. L'ascensore impiega due secondi per percorrere ciascun piano e stampa a video dei messaggi esplicativi, come nel seguente caso d'uso.

Attenzione: verrà valutato negativamente l'uso di attesa attiva.

Esempio d'uso:	Output dell'esempio d'uso:
Elevator e = new Elevator(10);	Elevator leaves floor 0
e. call(8);	Elevator stops at floor 3 (dopo 6 secondi)
e. call(5);	Elevator leaves floor 3
e. call(3);	Elevator stops at floor 4 (dopo 2 secondi)
e. call(4);	Elevator leaves floor 4
	Elevator stops at floor 5 (dopo 2 secondi)
	Elevator leaves floor 5
	Elevator stops at floor 8 (dopo 6 secondi)

177. (13 punti, 9 luglio 2009) La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A {
    public interface Convertible<T> {
        public T convert();
    }
    private Convertible<A> x = new B();
    private Iterable<A> y = new B(3);

    private Iterable<A> z = B.g(x);
    private Iterable<? extends B> t = B.g(B.b);
}
```

178. (20 punti, 9 luglio 2009) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ equals è un metodo statico della classe Object.
- ☐ ☐ L'interfaccia Cloneable è vuota.
- ☐ ☐ Uno dei metodi sort di Collections prende come argomento una Collection.
- ☐ ☐ Il metodo getClass di Object restituisce la classe effettiva di this.
- ☐ ☐ Qualunque oggetto può essere lanciato con throw.
- ☐ ☐ HashSet<Integer> è sottotipo di Iterable<Integer>.
- ☐ ☐ Il pattern Decorator si applica quando c'è un insieme prefissato di decorazioni possibili.
- ☐ ☐ Il pattern Iterator prevede un metodo per rimuovere l'ultimo elemento visitato dall'iteratore.
- ☐ ☐ Il metodo toString di Object rappresenta un'istanza del pattern Factory Method.
- ☐ ☐ Ridefinire il metodo equals tramite overriding rappresenta un'istanza del pattern Strategy.

179. **BoolExpr.** (33 punti, 25 febbraio 2008) La classe (o interfaccia) BoolExpr rappresenta un'espressione dell'algebra booleana (ovvero un circuito combinatorio). Il tipo più semplice di espressione è una semplice variabile, rappresentata dalla classe BoolVar, sottotipo di BoolExpr. Espressioni più complesse si ottengono usando gli operatori di tipo and, or e not, corrispondenti ad altrettante classi sottotipo di BoolExpr. Tutte le espressioni hanno un metodo eval che, dato il valore assegnato alle variabili, restituisce il valore dell'espressione. Si consideri *attentamente* il seguente caso d'uso.

<p>Esempio d'uso:</p> <pre> public static void main(String args[]) { BoolVar x = new BoolVar("x"); BoolVar y = new BoolVar("y"); BoolExpr notx = new BoolNot(x); BoolExpr ximpliesy = new BoolOr(notx, y); Map<BoolVar, Boolean> m = new HashMap<BoolVar, Boolean>() ; m.put(x, true); m.put(y, true); System.out.println(x.eval(m)); System.out.println(ximpliesy.eval(m)); m.put(y, false); System.out.println(ximpliesy.eval(m)); } </pre>	<p>Output dell'esempio d'uso:</p> <pre> true true false </pre>
--	--

180. (15 punti, 25 febbraio 2008) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public int f(int a, double b) { return 1 + f(8, 8); }
    public int f(float a, double b) { return 1 + f(7, b); }
    private int f(int a, float b) { return 1; }
}
class B extends A {
    public int f(int a, double b) { return 4; }
    public int f(double a, double b) { return 5; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa1 = new B();
        A alfa2 = new A();

        System.out.println(alfa1.f(1,2));
        System.out.println(alfa2.f(1,2));
        System.out.println(beta.f(1.0,2));
        System.out.println(8 | 1);
    }
}

```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

181. **MyFor.** (17 punti, 25 febbraio 2008) Implementare una classe `MyFor` in modo che, per tutti i numeri interi a , b e c , il ciclo:

```
for (Integer i: new MyFor(a, b, c)) { ... }
```

sia equivalente al ciclo:

```
for (Integer i=a; i<b ; i+=c) { ... }
```

182. (15 punti, 25 febbraio 2008) La seguente classe `A` fa riferimento ad una classe `B`. Implementare la classe `B` in modo che venga compilata correttamente e permetta la compilazione della classe `A`.

```

public class A {
    private B myb;

    private B f(B b) {
        myb = new B(true + "true");
        int x = b.confronta(myb);
        int y = myb.confronta(b);
        return myb.valore();
    }

    private Object zzz = B.z;
}

```

183. (22 punti, 25 febbraio 2008) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Una classe `final` può estendere un'altra classe.
- ☐ ☐ Un parametro formale può essere dichiarato `final`.
- ☐ ☐ La parola chiave `this` può essere usata per chiamare un costruttore.
- ☐ ☐ La classe `Thread` è astratta.
- ☐ ☐ `LinkedList<Integer>` è sottotipo di `LinkedList<Number>`.
- ☐ ☐ `LinkedList<Integer>` è sottotipo di `Iterable<Integer>`.
- ☐ ☐ L'istestazione di classe `class A<T,? extends T> extends T` è corretta.
- ☐ ☐ Un campo `protected` è visibile anche alle altre classi dello stesso pacchetto.
- ☐ ☐ Il pattern `Template Method` si può applicare solo ad algoritmi che fanno uso di determinate operazioni primitive.
- ☐ ☐ Per aggiungere funzionalità a una classe `A`, il pattern `Decorator` suggerisce di creare una sottoclasse di `A`.
- ☐ ☐ Nel pattern `Composite`, sia gli oggetti primitivi che composti implementano una stessa interfaccia.

184. **CommonDividers.** (30 punti, 20 luglio 2007) Implementare una classe `CommonDividers` che rappresenta tutti i divisori comuni di due numeri interi, forniti al costruttore. Su tale classe si deve poter iterare secondo il seguente caso d'uso. Dei 30 punti, 15 sono riservati a coloro che realizzeranno l'iteratore senza usare spazio aggiuntivo. Viene valutato positivamente l'uso di classi anonime laddove opportuno.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> for (Integer n: new CommonDividers(12, 60)) System.out.print(n + "_"); </pre>	<pre> 1 2 3 4 6 12 </pre>

185. (20 punti, 20 luglio 2007) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public static int x = 0;
    public A() { x++; }

    private int f(int a, double b) { return x; }
    public int f(int a, float b) { return x+5; }
    public int f(double a, double b) { return x+20; }
}

```

```

    public String toString() { return f(x, x) + ""; }
}
class B extends A {
    public int f(int a, float b) { return x-5; }
    public int f(int a, int b) { return x-10; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa1 = beta;
        A alfa2 = new A();

        System.out.println( alfa1 );
        System.out.println( alfa2 );
        System.out.println( beta );
        System.out.println( beta.f(4, 5.0) );
        System.out.println( 322 | 1 );
    }
}

```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

186. **ParkingLot.** (40 punti, 20 luglio 2007) Implementare una classe `ParkingLot`, che rappresenta un parcheggio con posti auto disposti secondo una griglia $m \times n$. Il costruttore prende come argomenti le dimensioni m ed n del parcheggio. Il metodo `carIn` aggiunge un veicolo al parcheggio e restituisce la riga e la colonna del posto assegnato al nuovo veicolo, oppure `null` se il parcheggio è pieno. Il metodo `carOut` prende come argomenti le coordinate di un veicolo che sta lasciando il parcheggio e restituisce il numero di secondi trascorsi dal veicolo nel parcheggio, oppure `null` se alle coordinate indicate non si trova alcun veicolo.

Suggerimento: utilizzare la classe `java.util.Date` per misurare il tempo.

Esempio d'uso:	Output:
<pre> ParkingLot p = new ParkingLot(10, 10); Pair<Integer> pos1 = p.carIn(); Pair<Integer> pos2 = p.carIn(); Thread.sleep(1000); int sec1 = p.carOut(pos1); Thread.sleep(1000); int sec2 = p.carOut(pos2); System.out.println("(" + pos1.getFirst() + ", " + pos1.getSecond() + "), " + sec1); System.out.println("(" + pos2.getFirst() + ", " + pos2.getSecond() + "), " + sec2); </pre>	<pre> (0, 0), 1 (0, 1), 2 </pre>

187. (15 punti, 20 luglio 2007) Individuare gli errori di *compilazione* nella seguente classe. Commentare brevemente ciascun errore e fornire una possibile correzione.

```

1 public class Errors {
2     private Errors e = null;
3     private Class<? extends String> c = String.getClass();
4
5     public Errors(Errors ee) { e = ee; }
6     public Errors() { this(this); }

```

```

7
8  public boolean f() {
9      Class<?> old_c = c;
10     c = Object.class;
11     return (old_c == c);
12 }
13 }

```

188. (20 punti, 20 luglio 2007) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ `OutOfMemoryError` è un'eccezione verificata.
 - ☐ ☐ Il blocco `try { ... } catch (Exception e)` cattura anche `NullPointerException`.
 - ☐ ☐ Una variabile locale può essere `private`.
 - ☐ ☐ Se `T` è una variabile di tipo, si può scrivere `new LinkedList<T>()`.
 - ☐ ☐ `HashSet<Integer>` è sottotipo di `Set<Integer>`.
 - ☐ ☐ Un metodo statico può essere `abstract`.
 - ☐ ☐ Un metodo non statico di una classe può chiamare un metodo statico della stessa classe.
 - ☐ ☐ Nel pattern `Decorator`, l'oggetto decoratore si comporta come l'oggetto da decorare.
 - ☐ ☐ Nel pattern `Decorator`, l'oggetto da decorare ha un metodo che aggiunge una decorazione.
 - ☐ ☐ Il metodo `Collections.sort` rappresenta un'istanza del pattern `Strategy`.
189. **Simulazione di ParkingLot.** (25 punti, 20 luglio 2007) Utilizzando la classe `ParkingLot` descritta nell'esercizio 3, scrivere un programma che simula l'ingresso e l'uscita di veicoli da un parcheggio. Un primo thread aggiunge un veicolo ogni secondo (a meno che il parcheggio non sia pieno). Un secondo thread, ogni due secondi, rimuove un veicolo dal parcheggio (a meno che il parcheggio non sia vuoto) e stampa a video il numero di secondi che tale veicolo ha trascorso nel parcheggio. Non ha importanza in che ordine i veicoli vengono rimossi dal parcheggio.
190. **Impianto e Apparecchio.** (30 punti, 27 marzo 2008) Si implementi una classe `Impianto` che rappresenta un impianto elettrico, e una classe `Apparecchio` che rappresenta un apparecchio elettrico collegabile ad un impianto. Un impianto è caratterizzato dalla sua potenza massima erogata (in Watt). Ciascun apparecchio è caratterizzato dalla sua potenza assorbita (in Watt). Per quanto riguarda la classe `Impianto`, il metodo `collega` collega un apparecchio a questo impianto, mentre il metodo `potenza` restituisce la potenza attualmente assorbita da tutti gli apparecchi *collegati* all'impianto ed *accesi*.

I metodi `on` e `off` di ciascun apparecchio accendono e spengono, rispettivamente, questo apparecchio. Se, accendendo un apparecchio col metodo `on`, viene superata la potenza dell'impianto a cui è collegato, deve essere lanciata una eccezione.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> Apparecchio tv = new Apparecchio(150); Apparecchio radio = new Apparecchio(30); Impianto i = new Impianto(3000); i.collega(tv); i.collega(radio); System.out.println(i.potenza()); tv.on(); System.out.println(i.potenza()); radio.on(); System.out.println(i.potenza()); </pre>	<pre> 0 150 180 </pre>

191. (20 punti, 27 marzo 2008) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    private int f(double a, int b, A c) { return 1; }
    public int f(double a, float b, A c) { return 20; }
    public int f(long a, float b, B c) { return 10; }
}
class B extends A {
    public int f(double a, float b, A c) { return 30; }
    public int f(int a, int b, B c) { return 40; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;

        System.out.println(alfa.f(1,2, alfa));
        System.out.println(alfa.f(1,2, null));
        System.out.println(beta.f(1,2, beta));
        System.out.println(beta.f(1.0,2, beta));
        System.out.println(1234 & 1234);
    }
}
```

- Indicare l'output del programma.
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
192. **DelayIterator.** (15 punti, 27 marzo 2008) Implementare un metodo statico `delayIterator` che prende come argomenti un iteratore i ed un numero intero n , e restituisce un nuovo iteratore dello stesso tipo di i , che restituisce gli stessi elementi di i , ma in cui ogni elemento viene restituito (dal metodo `next`) dopo un ritardo di n secondi. Viene valutato positivamente l'uso di classi anonime.

Si ricordi che nella classe `Thread` è presente il metodo:

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

193. (15 punti, 27 marzo 2008) La seguente classe `A` fa riferimento ad una classe `B`. Implementare la classe `B` in modo che venga compilata correttamente e permetta la compilazione della classe `A`.

```
public class A extends B {
    public A(int x) {
        super(x-1, x / 2.0);
    }
    public A(double inutile) { }

    private void stampa(String s) {
        if (s == null) throw new B(s);
        else System.out.println(s);
    }
}
```

194. (20 punti, 27 marzo 2008) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Un campo `protected` è visibile anche alle altre classi dello stesso pacchetto.
- ☐ ☐ La parola chiave `synchronized` si può applicare anche ad un campo.
- ☐ ☐ L'istruzione `Number[] n = new Integer[10];` è corretta.
- ☐ ☐ L'istruzione `LinkedList<Number> l = new LinkedList<Integer>();` è corretta.
- ☐ ☐ Si può effettuare l'overriding di un metodo statico.
- ☐ ☐ Un metodo statico può contenere una classe locale.
- ☐ ☐ `Thread` è un'interfaccia della libreria standard.
- ☐ ☐ Il pattern `Composite` prevede che sia gli oggetti primitivi sia quelli composti implementino una stessa interfaccia.
- ☐ ☐ Nel pattern `Observer`, un oggetto può essere osservato da al più un osservatore.
- ☐ ☐ Nell'architettura `Model-View-Controller`, solo i controller dovrebbero modificare i modelli.

195. **Aereo.** (25 punti, 17 settembre 2007) Si implementi una classe `Aereo`. Ogni aereo si può trovare in ogni istante di tempo in uno dei seguenti quattro stati: in fase di decollo, in fase di crociera, in fase di atterraggio, atterrato. I quattro metodi `decollo`, `crociera`, `atterraggio`, `atterrato` cambiano lo stato dell'aereo. Questi metodi devono sollevare un'eccezione nuova, definita da voi, se non vengono chiamati nell'ordine giusto. Infine, il metodo `nvoli` restituisce il numero di voli completati dall'aereo fino a quel momento.

196. (25 punti, 17 settembre 2007) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    private int f(int a, double b, double c) { return 1; }
    public int f(int a, float b, double c) { return 10; }
    public int f(double a, double b, double c) { return 20; }
}
class B extends A {
    public int f(int a, float b, int c) { return 15; }
    public int f(int a, float b, double c) { return 25; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa1 = beta;
        A alfa2 = new A();

        System.out.println(alfa1.f(1,2,3));
        System.out.println(alfa2.f(1,2,3));
        System.out.println(beta.f(1,2,3));
        System.out.println(beta.f(1.0,2,3));
        System.out.println(7 / 2);
    }
}
```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

197. **Selector.** (35 punti, 17 settembre 2007) L'interfaccia parametrica `Selector` prevede un metodo `select` che restituisce un valore booleano per ogni elemento del tipo parametrico.

```
public interface Selector<T> {
    boolean select(T x);
}
```

Implementare una classe `SelectorIterator` che accetta una collezione e un selettore dello stesso tipo, e permette di iterare sugli elementi della collezione per i quali il selettore restituisce `true`.

Esempio d'uso:	Output dell'esempio d'uso:
<pre>Integer[] a = { 1, 2, 45, 56, 343, 22, 12, 7, 56}; List<Integer> l = Arrays.asList(a); Selector<Integer> pari = new Selector<Integer>() { public boolean select(Integer n) { return (n % 2) == 0; } }; for (Integer n: new SelectorIterator<Integer>(l, pari)) System.out.print(n + " ");</pre>	<pre>2 56 22 12 56</pre>

198. **FunnyOrder.** (20 punti, 17 settembre 2007) Determinare l'output del seguente programma e descrivere brevemente l'ordinamento dei numeri interi definito dalla classe `FunnyOrder`.

```
public class FunnyOrder implements Comparable<FunnyOrder> {
    private int val;
    public FunnyOrder(int n) { val = n; }
    public int compareTo(FunnyOrder x) {
        if (val%2 == 0 && x.val%2 != 0) return -1;
        if (val%2 != 0 && x.val%2 == 0) return 1;
        if (val < x.val) return -1;
        if (val > x.val) return 1;
        return 0;
    }
    public static void main(String[] args) {
        List<FunnyOrder> l = new LinkedList<FunnyOrder>();
        l.add(new FunnyOrder(16));
        l.add(new FunnyOrder(3));
        l.add(new FunnyOrder(4));
        l.add(new FunnyOrder(10));
        l.add(new FunnyOrder(2));
        Collections.sort(l);
        for (FunnyOrder f: l)
            System.out.println(f.val + " ");
    }
}
```

199. (20 punti, 17 settembre 2007) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Se un metodo contiene una dichiarazione `throws`, ogni metodo che ne faccia l'overriding deve contenere una dichiarazione `throws`.
- ☐ ☐ L'eccezione `ArrayIndexOutOfBoundsException` non può essere catturata.
- ☐ ☐ Un metodo statico può accedere ai campi statici della classe.
- ☐ ☐ La classe `Class` è astratta.
- ☐ ☐ Una classe non parametrica può implementare `Collection<String>`.
- ☐ ☐ Nel binding dinamico, la lista delle firme candidate può essere vuota.
- ☐ ☐ Un metodo `synchronized` di un oggetto può essere chiamato da un solo thread per volta.
- ☐ ☐ Il pattern `Iterator` prevede un metodo per far avanzare di una posizione l'iteratore.

- □ Nel pattern **Observer**, gli osservatori devono contenere un riferimento all'oggetto osservato.
- □ Il pattern **Strategy** permette di fornire versioni diverse di un algoritmo.

200. **Genealogia.** (35 punti, 26 aprile 2007) Nell'ambito di un programma di genealogia, si implementi la classe (o interfaccia) **Person** e le sottoclassi **Man** e **Woman**, con le seguenti caratteristiche. Una persona è dotata di nome e cognome. Il metodo **addChild** di **Person** prende una persona x come argomento e segnala che x è figlia di **this**. Il metodo **marries** di **Person** prende una persona x come argomento e segnala che x è sposata con **this**. Il metodo **marries** lancia un'eccezione se x è dello stesso genere di **this**. Il metodo statico **areSiblings** prende come argomenti due persone x e y e restituisce vero se x ed y sono fratelli o sorelle e falso altrimenti.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> Person a = new Man("Mario", "Rossi"); Person b = new Woman("Luisa", "Verdi"); Person c = new Man("Luca", "Rossi"); Person d = new Woman("Anna", "Rossi"); Person e = new Woman("Daniela", "Rossi"); a.marries(b); a.addChild(c); b.addChild(d); c.addChild(e); System.out.println(Person.areSiblings(a, b)); System.out.println(Person.areSiblings(c, d)); </pre>	<pre> so: false true </pre>

201. **AncestorIterator.** (25 punti, 26 aprile 2007) Con riferimento all'Esercizio 1, definire una classe **AncestorIterator** che itera su tutti gli antenati conosciuti di una persona, in ordine arbitrario. Ad esempio, si consideri il seguente caso d'uso, che fa riferimento alle persone a,b,c,d ed e dell'Esercizio 1.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> Iterator i = new AncestorIterator(e); while (i.hasNext()) { System.out.println(i.next()); } </pre>	<pre> Luca Rossi Mario Rossi Luisa Verdi </pre>

Dei 25 punti, 10 sono riservati a coloro che implementeranno **AncestorIterator** come classe interna di **Person**. In tal caso, il primo rigo dell'esempio d'uso diventa:

```
Iterator i = e.new AncestorIterator();
```

Suggerimento: si ricorda che se **B** è una classe interna di **A**, all'interno di **B** il riferimento implicito all'oggetto di tipo **A** si chiama **A.this**.

202. (25 punti, 26 aprile 2007) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

abstract class A {
    public abstract String f(A other, int n);
    public String f(B other, long n) { return "A2:" + n; }
}
class B extends A {
    public String f(A other, int n) { return "B1:" + n; }
    private String f(C other, long n) { return "B2:" + n; }
}

```

```

class C extends B {
    public String f(A other, long n) { return "C1:" + n; }
    public String f(C other, long n) { return "C2:" + n; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(15 & 7);
        System.out.println(alfa.f(alfa, 4));
        System.out.println(alfa.f(beta, 4L));
        System.out.println(beta.f(gamma, 4L));
        System.out.println(gamma.f(gamma, 3L));
    }
}

```

- Indicare l'output del programma. (20 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (5 punti)

203. (15 punti, 26 aprile 2007) Individuare gli errori di compilazione nel seguente programma. Commentare brevemente ciascun errore e fornire una possibile correzione.

```

1 public class Errors {
2     private static int sval = 7;
3     private int val = sval;
4
5     public Errors() { super(); }
6
7     private class A {
8         private A(int n) { val += n; }
9     }
10    private class B extends A {
11        B() { val = sval; }
12    }
13
14    public static void main(String[] args) {
15        Errors t = new Errors;
16        A a = t.new A(5);
17        B b = a.new B();
18    }
19 }

```

204. **Rational.** (34 punti, 29 giugno 2007)

- (18 punti) Si implementi una classe `Rational` che rappresenti un numero razionale in maniera esatta. Il costruttore accetta numeratore e denominatore. Se il denominatore è negativo, viene lanciata una eccezione. Il metodo `plus` prende un altro `Rational` x come argomento e restituisce la somma di `this` e x . Il metodo `times` prende un altro `Rational` x come argomento e restituisce il prodotto di `this` e x .
- (9 punti) La classe deve assicurarsi che numeratore e denominatore siano sempre ridotti ai minimi termini. (Suggerimento: la minimizzazione della frazione può essere compito del costruttore)
- (7 punti) La classe deve implementare l'interfaccia `Comparable<Rational>`, in base al normale ordinamento tra razionali.

Esempio d'uso: <pre> Rational n = new Rational(2,12); // due dodicesimi Rational m = new Rational(4,15); // quattro quindicesimi Rational o = n.plus(m); Rational p = n.times(m); System.out.println(n); System.out.println(o); System.out.println(p); </pre>	Output dell'esempio d'uso: <pre> 1/6 13/30 2/45 </pre>
---	---

205. (16 punti, 29 giugno 2007) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public int f(int x, A a) { return 0; }
    public int f(double x, B b) { return 7; }
    public int f(double x, A a) { return 10; }
}
class B extends A {
    public int f(int x, B b) { return f(2.0 * x, b) + 1; }
    public int f(double x, B b) { return 20; }
    public int f(double x, A a) { return f((int) x, a) + 1; }
}
public class Test2 {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;

        System.out.println(alfa.f(3.0, beta));
        System.out.println(alfa.f(3.0, alfa));
        System.out.println(beta.f(3, beta));
        System.out.println(beta.f(3, alfa));
    }
}

```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

206. **Polinomio su un campo generico.** (40 punti, 29 giugno 2007) Un *campo* (field) è una struttura algebrica composta da un insieme detto supporto, dalle due operazioni binarie di somma e prodotto, e dai due elementi neutri, per la somma e per il prodotto rispettivamente. La seguente interfaccia rappresenta un campo con supporto `T`:

```

public interface Field<T> {
    T plus(T x, T y); // la somma
    T times(T x, T y); // il prodotto
    T getOne(); // restituisce l'elemento neutro per il prodotto
    T getZero(); // restituisce l'elemento neutro per la somma
}

```

- (10 punti) Implementare una classe `DoubleField` che implementi `Field<Double>`.
- (30 punti) Implementare una classe `Polynomial` che rappresenti un polinomio con coefficienti in un dato campo. Il costruttore accetta un array di coefficienti e il campo sul quale interpretare i coefficienti. Il metodo `eval` restituisce il valore del polinomio per un dato valore della variabile.

Esempio d'uso:	Output:
<pre>Double[] d = { 2.0, 3.0, 1.0 }; // 2 + 3x + x^2 Polynomial<Double> p = new Polynomial<Double>(d, new DoubleField()); System.out.println(p.eval(3.0)); System.out.println(p.eval(2.0));</pre>	<pre>20.0 12.0</pre>

207. (15 punti, 29 giugno 2007) Individuare gli errori di compilazione nella seguente classe. Commentare brevemente ciascun errore e fornire una possibile correzione.

```

1 public class Errors {
2     private static int num = 7;
3     private Integer z = 8;
4     Map<Integer, Errors> m = new Map<Integer, Errors>();
5
6     public Errors() { }
7
8     private static class A {
9         private A() { num += z; }
10    }
11    private void f() {
12        m.put(7, new Errors() { public int g() { return 0; } });
13    }
14
15    public static final A a = new A();
16 }
```

208. (20 punti, 29 giugno 2007) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Una classe può avere più metodi pubblici con lo stesso nome e lo stesso tipo restituito.
- ☐ ☐ Si può effettuare l'overriding di un costruttore.
- ☐ ☐ I costruttori possono sollevare eccezioni.
- ☐ ☐ Una classe **abstract** può avere campi.
- ☐ ☐ `LinkedList<Integer>` è sottotipo di `LinkedList<Number>`.
- ☐ ☐ Una classe anonima può avere costruttore.
- ☐ ☐ Un metodo pubblico di una classe può chiamare un metodo privato della stessa classe.
- ☐ ☐ Il pattern **Iterator** prevede che un iteratore abbia un metodo **remove**.
- ☐ ☐ Nell'architettura MVC, i controller non devono comunicare direttamente con i modelli.
- ☐ ☐ Nel pattern **Strategy**, si suggerisce di usare una classe per rappresentare un'algoritmo.

209. **Highway.** (25 punti, 29 giugno 2007) Implementare una classe **Highway**, che rappresenti un'autostrada a senso unico. Il costruttore accetta la lunghezza dell'autostrada in chilometri. Il metodo **insertCar** prende un intero x come argomento ed aggiunge un'automobile al chilometro x . L'automobile inserita percorrerà l'autostrada alla velocità di un chilometro al minuto, (60 km/h) fino alla fine della stessa. Il metodo **nCars** prende un intero x e restituisce il numero di automobili presenti al chilometro x . Il metodo **progress** simula il passaggio di 1 minuto di tempo (cioè fa avanzare tutte le automobili di un chilometro).

Si supponga che thread multipli possano accedere allo stesso oggetto **Highway**.

Dei 25 punti, 8 sono riservati a coloro che implementeranno **progress** in tempo indipendente dal numero di automobili presenti sull'autostrada.

Esempio d'uso:	Output:
<pre> Highway h = new Highway(10); h.insertCar(3); h.insertCar(3); h.insertCar(5); System.out.println(h.nCars(4)); h.progress(); System.out.println(h.nCars(4)); </pre>	<pre> 0 2 </pre>

210. **Recipe.** (30 punti, 30 gennaio 2008) Si implementi una classe **Recipe** che rappresenta una ricetta. Il costruttore accetta il nome della ricetta. Il metodo **setDescr** imposta la descrizione della ricetta. Il metodo **addIngr** aggiunge un ingrediente alla ricetta, prendendo come primo argomento la quantità (anche frazionaria) dell'ingrediente, per una persona, e come secondo argomento una stringa che contiene l'unità di misura e il nome dell'ingrediente. Se un ingrediente è difficilmente misurabile, si imposterà la sua quantità a zero, e verrà visualizzato come "q.b." ("quanto basta"). Il metodo **toString** prende come argomento il numero di coperti n e restituisce una stringa che rappresenta la ricetta, in cui le quantità degli ingredienti sono state moltiplicate per n .

Esempio d'uso:	Output dell'esempio d'uso:
<pre> Recipe r = new Recipe("Spaghetti_aglio_e_olio"); r.addIngr(100, "grammi_di_spaghetti"); r.addIngr(2, "cucchiaini_d'olio_d'oliva"); r.addIngr(1, "spicchi_d'aglio"); r.addIngr(0, "sale"); r.setDescr("Mischiare_tutti_gli_ingredienti_e_servire."); System.out.println(r.toString(4)); </pre>	<pre> Spaghetti aglio e olio Ingredienti per 4 persone: 400 grammi di spaghetti 8 cucchiaini d'olio d'oliva 4 spicchi d'aglio q.b. sale Preparazione: Mischiare tutti gli ingredienti e servire. </pre>

211. (15 punti, 30 gennaio 2008) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public int f(int a, float b, double c) { return 1; }
    public int f(double a, double b, double c) { return 2; }
    private int f(int a, double b, double c) { return 3; }
}
class B extends A {
    public int f(int a, float b, double c) { return 4; }
    public int f(int a, float b, int c) { return 5; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa1 = beta;
        A alfa2 = new A();

        System.out.println(alfa1.f(1,2,3));
        System.out.println(alfa2.f(1,2,3));
        System.out.println(beta.f(1.0,2,3));
        System.out.println(177 & 2);
    }
}

```

- Indicare l'output del programma.

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
212. **Sorter.** (15 punti, 30 gennaio 2008) Implementare una classe parametrica `Sorter`, con un solo metodo `check`. Il metodo `check` confronta l'oggetto che riceve come argomento con quello che ha ricevuto alla chiamata precedente, o con quello passato al costruttore se si tratta della prima chiamata a `check`. Il metodo restituisce -1 se il nuovo oggetto è più piccolo del precedente, 1 se il nuovo oggetto è più grande del precedente e 0 se i due oggetti sono uguali. Per effettuare i confronti, `Sorter` si basa sul fatto che il tipo usato come parametro implementi l'interfaccia `Comparable`.

Esempio d'uso:	Output dell'esempio d'uso:
<code>Sorter<Integer> s = new Sorter<Integer>(7);</code>	-1
<code>System.out.println(s.check(4));</code>	-1
<code>System.out.println(s.check(1));</code>	1
<code>System.out.println(s.check(6));</code>	0
<code>System.out.println(s.check(6));</code>	

213. (20 punti, 30 gennaio 2008) La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A {
    private B myb;

    private int f(B b) {
        A x = B.copia(b);
        myb = B.copia(77);
        double d = myb.g();
        return myb.g();
    }

    private int x = B.x;
}
```

214. (20 punti, 30 gennaio 2008) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Un campo `protected` è visibile anche alle altre classi dello stesso pacchetto.
- ☐ ☐ Il costrutto `catch A` cattura le eccezioni di tipo A e delle sue sottoclassi.
- ☐ ☐ Un'interfaccia può avere un campo statico `final`.
- ☐ ☐ Un costruttore può essere dichiarato `final`.
- ☐ ☐ Applicato ad un metodo, il modificatore `final` impedisce sia l'overloading che l'overriding.
- ☐ ☐ Una classe non parametrica può implementare `Comparable<Integer>`.
- ☐ ☐ Un costruttore di una classe non parametrica può avere un parametro di tipo.
- ☐ ☐ La scelta del layout in un pannello Swing/AWT è un esempio del pattern **Strategy**.
- ☐ ☐ Per aggiungere funzionalità a una classe A, il pattern **Decorator** suggerisce di creare una sottoclasse di A.

- □ Nel pattern **Composite**, i client devono poter distinguere tra un oggetto primitivo e un oggetto composito.

215. **FallingBody**. (25 punti, 15 Settembre 2006) Nel contesto di un programma di simulazione per la cinematica, si implementi una classe **FallingBody** che rappresenta un corpo puntiforme dotato di massa, che cade soggetto solo alla forza di gravità terrestre. Il costruttore della classe prende come argomento la massa del corpo e la sua altezza iniziale. Si supponga che tutte le grandezze siano espresse in unità tra loro omogenee (altezza in metri, velocità in metri al secondo, etc.). Il metodo **progress** simula il passaggio di un dato numero di secondi. Il metodo **toString** va ridefinito in modo da mostrare l'altezza dal suolo e la velocità corrente del corpo. Non deve essere possibile creare sottoclassi di **FallingBody**.

Si supponga che l'accelerazione di gravità sia pari a $10\frac{m}{s^2}$. Si ricordano le equazioni del moto uniformemente accelerato.

$$v = v_0 + at; \qquad s = s_0 + v_0t + \frac{1}{2}at^2.$$

Esempio d'uso:	Output dell'esempio d'uso:
<pre>// Corpo di 2 kili , ad un'altezza di 20 metri. FallingBody b = new FallingBody(2, 20) ; System.out.println(b); b.progress(1); System.out.println(b); b.progress(1); System.out.println(b); b.progress(7); System.out.println(b);</pre>	<pre>altezza: 20.0, velocita': 0.0 altezza: 15.0, velocita': 10.0 altezza: 0.0, velocita': 0.0 altezza: 0.0, velocita': 0.0</pre>

216. (25 punti, 15 Settembre 2006) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
public class A {
    public boolean equals(A other) {
        System.out.println("in_A:");
        return true;
    }
}
public class B extends A {
    public boolean equals(A other) {
        System.out.println("in_B:");
        return true;
    }
}
public class C extends B {
    public boolean equals(Object other) {
        System.out.println("in_C:");
        return true;
    }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;
        System.out.println(alfa.equals(beta));
        System.out.println(beta.equals(alfa));
        System.out.println(gamma.equals(alfa));
        System.out.println(gamma.equals( new String("ciao") ));
```

```

        System.out.println(15 & 1);
    }
}

```

- Indicare l'output del programma. (20 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (5 punti)

217. **TreeType.** (40 punti, 15 Settembre 2006) Implementare le classi `TreeType` e `Tree`. `TreeType` rappresenta un tipo di albero (pino, melo, etc.), mentre `Tree` rappresenta un particolare esemplare di albero. Ogni `TreeType` è caratterizzato dal suo nome. Ogni `Tree` ha un tipo base ed eventualmente degli innesti di altri tipi di alberi. Il metodo `addGraft` di `Tree` aggiunge un innesto ad un albero, purchè non sia dello stesso tipo dell'albero stesso. Il metodo `getCounter` di `Tree` restituisce il numero di alberi che sono stati creati. Il metodo `getCounter` di `TreeType` restituisce il numero di alberi di quel tipo che sono stati creati. (32 punti)

Ridefinire il metodo `clone` di `Tree`, facendo attenzione ad eseguire una copia profonda laddove sia necessario. (8 punti)

Esempio d'uso:	Output dell'esempio d'uso:
<pre> TreeType melo = new TreeType("melo"); TreeType pero = new TreeType("pero"); ; Tree unMelo = new Tree(melo); Tree unAltroMelo = new Tree(melo); unAltroMelo.addGraft(pero); unAltroMelo.addGraft(pero); System.out.println("Sono stati creati_" + melo.getCounter() + "_meli_fino_ a_questo_momento."); System.out.println("Sono stati creati_" + Tree.getCounter() + "_alberi_fino_ a_questo_momento."); System.out.println(unAltroMelo); unAltroMelo.addGraft(melo); </pre>	<pre> Sono stati creati 2 meli fino a questo momento. Sono stati creati 2 alberi fino a questo momento. tipo: melo innesti: pero Exception in thread "main": java.lang.RuntimeException </pre>

218. (20 punti, 15 Settembre 2006) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti, assenza di risposta 0 punti. Se il totale è negativo, vale 0.

Vero Falso

- | | | |
|--------------------------|--------------------------|---|
| <input type="checkbox"/> | <input type="checkbox"/> | L'interfaccia <code>Iterable</code> ha un parametro di tipo. |
| <input type="checkbox"/> | <input type="checkbox"/> | La classe <code>String</code> ha un parametro di tipo. |
| <input type="checkbox"/> | <input type="checkbox"/> | Il tipo jolly "?" è un parametro attuale di tipo. |
| <input type="checkbox"/> | <input type="checkbox"/> | Una classe può avere più di un parametro di tipo. |
| <input type="checkbox"/> | <input type="checkbox"/> | L'interfaccia <code>List</code> estende <code>Collection</code> . |
| <input type="checkbox"/> | <input type="checkbox"/> | Le eccezioni derivate da <code>RuntimeException</code> non sono verificate. |
| <input type="checkbox"/> | <input type="checkbox"/> | Gli oggetti di tipo <code>String</code> sono immutabili. |
| <input type="checkbox"/> | <input type="checkbox"/> | Dato un insieme di firme di metodi, non sempre ce ne è una più specifica di tutte le altre. |
| <input type="checkbox"/> | <input type="checkbox"/> | L' <i>early binding</i> è svolto dal programmatore. |

□ □ Il *late binding* è svolto dalla Java Virtual Machine.

219. **SuperclassIterator.** (25 punti, 15 Settembre 2006) Implementare una classe **SuperclassIterator** che rappresenta un iteratore su tutte le superclassi di un oggetto dato, a partire dalla classe stessa dell'oggetto fino ad arrivare ad **Object**.

Ad esempio, nell'ambito della tradizionale gerarchia formata dalle classi **Employee** e **Manager**, si consideri il seguente caso d'uso.

<p>Esempio d'uso:</p> <pre> Iterator <Class<?>> i = new SuperclassIterator(new Manager("Franco")); while (i.hasNext()) System.out.println(i.next()); </pre>	<p>Output dell'esempio d'uso:</p> <pre> class Manager class Employee class java.lang.Object </pre>
--	--

220. (15 punti, 15 Settembre 2006) Individuare e descrivere sinteticamente gli eventuali errori nel seguente programma. Il programma dovrebbe lanciare un nuovo thread che stampa gli interi da 0 a 9.

```

1  class Test extends Runnable {
2      private Thread thread;
3
4      public Test() {
5          thread = new Thread();
6      }
7
8      public run() {
9          int i = 0;
10         for (i=0; i<10 ;i++)
11             System.out.println("_i_=_ " + i);
12     }
13
14     public static void main(String args[]) {
15         Test t = new Test();
16         t.start();
17     }
18 }

```

221. **Moto accelerato.** (25 punti, 17 Luglio 2006) Nel contesto di un programma di simulazione per la cinematica, si implementi una classe **Body** che rappresenta un corpo puntiforme dotato di massa, che si sposta lungo una retta. Il costruttore della classe prende come argomento la massa del corpo. Il corpo si suppone inizialmente in quiete alla coordinata 0. Il metodo **setForce** imposta il valore di una forza che viene applicata al corpo. Si supponga che tutte le grandezze siano espresse in unità tra loro omogenee (posizione in metri, velocità in metri al secondo, forza in Newton, etc.). Il metodo **progress** simula il passaggio di un dato numero di secondi, andando ad aggiornare la posizione del corpo. Il metodo **toString** va ridefinito in modo da mostrare la posizione e la velocità corrente del corpo.

Si ricordano le equazioni del moto uniformemente accelerato.

$$F = ma; \quad v = v_0 + at; \quad s = s_0 + v_0 t + \frac{1}{2} at^2.$$

Esempio d'uso:	Output dell'esempio d'uso:
<pre> Body b = new Body(20); b.setForce(40); System.out.println(b); b.progress(1); System.out.println(b); b.progress(2); System.out.println(b); b.setForce(-100); b.progress(2); System.out.println(b); </pre>	<pre> posizione: 0.0, velocita': 0.0 posizione: 1.0, velocita': 2.0 posizione: 9.0, velocita': 6.0 posizione: 11.0, velocita': -4.0 </pre>

222. (25 punti, 17 Luglio 2006) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

public class A {
    public boolean equals(A other) {
        System.out.println("in_A:");
        return true;
    }
}
public class B extends A { }
public class C extends A {
    public boolean equals(Object other) {
        System.out.println("in_C:");
        return false;
    }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;
        System.out.println(alfa.equals(beta));
        System.out.println(gamma.equals(beta));
        System.out.println(beta.equals(alfa));
        System.out.println(beta.equals((Object) alfa));
        System.out.println("true" + true);
    }
}

```

- Indicare l'output del programma. (20 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (5 punti)

223. **Spartito.** (40 punti, 17 Luglio 2006) Nel contesto di un software per la composizione musicale, si implementi una classe **Nota**, e una classe **Spartito**. Ciascuna nota ha un nome e una durata. La durata può essere soltanto di 1, 2, oppure 4 unità di tempo (semiminima, minima oppure semibreve). Uno spartito è una sequenza di note, tale che più note possono cominciare (o terminare) nello stesso istante. Il metodo `add` della classe **Spartito** prende come argomento una nota ed un istante di tempo t , ed aggiunge la nota allo spartito, a partire dal tempo t . Quando si itera su uno spartito, ad ogni chiamata a `next` viene restituito l'insieme di note presenti nell'unità di tempo corrente.

Implementare tutti i metodi necessari a rispettare il seguente caso d'uso.

```

public static void main(String[] x) {
    Spartito fuga = new Spartito();
    fuga.add(new Nota("Do", 4), 0);
    fuga.add(new Nota("Mi", 1), 0);
}

```

```
fuga.add(new Nota("Mib", 2), 1);
fuga.add(new Nota("Sol", 2), 2);

for (Set<Nota> accordo : fuga)
    System.out.println(accordo);
}
```

Output del codice precedente:

```
Do, Mi
Do, Mib
Do, Mib, Sol
Do, Sol
```

224. (20 punti, 17 Luglio 2006) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti, assenza di risposta 0 punti. Se il totale è negativo, vale 0.

Vero Falso

- ☐ ☐ Una interfaccia può avere metodi statici.
- ☐ ☐ Un campo final pubblico di una classe non viene ereditato.
- ☐ ☐ LinkedList implementa Collection.
- ☐ ☐ Integer è sottoclasse di Number.
- ☐ ☐ Java è un linguaggio funzionale.
- ☐ ☐ OutOfMemoryError è una eccezione non verificata.
- ☐ ☐ Boolean è sottoclasse di Number.
- ☐ ☐ Boolean è sottoclasse di Object.
- ☐ ☐ sleep è un metodo statico di Runnable.
- ☐ ☐ "double d = 3;" è una istruzione corretta.

225. **TwoSteps.** (25 punti, 17 Luglio 2006) Implementare un metodo statico **twoSteps** che accetta come argomento un iteratore e restituisce un iteratore dello stesso tipo, che compie due passi per ogni chiamata a **next**.

Come esempio, si consideri il seguente caso d'uso.

Esempio d'uso:	Output dell'esempio d'uso:
<pre>List<Integer> l = new LinkedList<Integer>(); l.add(3); l.add(5); l.add(7); l.add(9); Iterator<Integer> iter1 = twoSteps(l.iterator()); System.out.println("Iterazione_1:"); System.out.println(iter1.next()); System.out.println(iter1.next()); Iterator<Integer> iter2 = twoSteps(l.iterator()); System.out.println("Iterazione_2:"); while (iter2.hasNext()) System.out.println(iter2.next());</pre>	<pre>Iterazione 1: 3 7 Iterazione 2: 3 7</pre>

226. (15 punti, 17 Luglio 2006) Individuare e descrivere sinteticamente gli eventuali errori nel seguente programma.

```

1  class Test {
2      Collection<?> c;
3
4      public int f(Collection<? extends Number> c) {
5          return c.size();
6      }
7
8      public void g(Set<? extends Number> c) {
9          this.c = c;
10     }
11
12     private <T extends Number> T myAdd(T x) {
13         c.add(x);
14         return x;
15     }
16
17     public static void main(String args[]) {
18         Test t = new Test();
19
20         t.f(new LinkedList<Integer>());
21         t.g(new ArrayList<Integer>());
22         t.myAdd(new Double(2.0));
23     }
24 }

```

227. **Inventory.** (30 punti, 23 febbraio 2007) Definire una classe parametrica `Inventory<T>` che rappresenta un inventario di oggetti di tipo `T`. Il costruttore senza argomenti crea un inventario vuoto. Il metodo `add` aggiunge un oggetto di tipo `T` all'inventario. Il metodo `count` prende come argomento un oggetto di tipo `T` e restituisce il numero di oggetti uguali all'argomento presenti nell'inventario. Infine, il metodo `getMostCommon` restituisce l'oggetto di cui è presente il maggior numero di esemplari.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> Inventory<Integer> a = new Inventory<Integer>(); Inventory<String> b = new Inventory<String>(); a.add(7); a.add(6); a.add(7); a.add(3); b.add("ciao"); b.add("allora?"); b.add("ciao_ciao"); b.add("allora?"); System.out.println(a.count(2)); System.out.println(a.count(3)); System.out.println(a.getMostCommon()); System.out.println(b.getMostCommon()); </pre>	<pre> 0 1 7 allora? </pre>

228. **Primes.** (25 punti, 23 febbraio 2007) Definire una classe `Primes` che rappresenta l'insieme dei numeri primi. Il campo statico `iterable` fornisce un oggetto su cui si può iterare, ottenendo l'elenco di tutti i numeri primi. Non deve essere possibile per un'altra classe creare oggetti di tipo `Primes`.

Suggerimento: `Primes` potrebbe implementare sia `Iterator<Integer>` che `Iterable<Integer>`. In tal caso, il campo `iterable` potrebbe puntare ad un oggetto di tipo `Primes`.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> for (Integer i: Primes.iterable) { if (i > 20) break; System.out.println(i); } </pre>	<pre> 1 3 5 7 11 13 17 19 </pre>

229. (25 punti, 23 febbraio 2007) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

public abstract class A {
    public abstract String f(A other, int n);
    public String f(A other, long n)    { return "A:" + n; }
}
public class B extends A {
    public String f(A other, int n)      { return "B1:" + n; }
    public String f(Object other, int n) { return "B2:" + n; }
    public String f(B other, long n)    { return "B3:" + n; }
}
public class C extends B {
    public String f(B other, long n)      { return "C1:" + n; }
    public String f(C other, int n)      { return "C2:" + n; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta  = gamma;
        A alfa  = gamma;
        System.out.println(alfa.f(beta, 4));
        System.out.println(alfa.f(beta, 4L));
        System.out.println(beta.f( (Object) alfa, 4));
        System.out.println(gamma.f(gamma, 3));
    }
}

```

- Indicare l'output del programma. (20 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (5 punti)

230. (20 punti, 23 febbraio 2007) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti, assenza di risposta 0 punti. Se il totale è negativo, vale 0.

Vero Falso

- ☐ ☐ Una variabile locale di un metodo può essere dichiarata **static**.
- ☐ ☐ Una classe astratta può implementare un'interfaccia.
- ☐ ☐ Un'interfaccia può avere un costruttore.
- ☐ ☐ Una classe deve necessariamente implementare un costruttore.
- ☐ ☐ Tutte le classi derivano da (cioè sono sottoclassi di) `Class`.
- ☐ ☐ Tutte le eccezioni derivano da `Exception`.
- ☐ ☐ `clone` è un metodo pubblico di `Object`.

- ☐ ☐ Se `x` è un riferimento ad un `Employee`, l'istruzione "`x instanceof Object`" restituisce `false`.
- ☐ ☐ Se un thread sta eseguendo un metodo `synchronized` di un oggetto, nessun altro thread può eseguire i metodi di quell'oggetto.
- ☐ ☐ Una variabile locale di un metodo può essere dichiarata `private`.

231. **Moto bidimensionale.** (30 punti, 26 Giugno 2006) Nel contesto di un programma di simulazione per la cinematica, si implementi una classe `Body` che rappresenta un corpo puntiforme dotato di posizione nel piano cartesiano e di velocità. Il costruttore della classe prende come argomento le coordinate alle quali si trova inizialmente il corpo; il corpo si suppone inizialmente in quiete. Il metodo `setSpeed` prende il valore della velocità lungo i due assi di riferimento. Si supponga che la posizione sia espressa in metri e la velocità in metri al secondo. Il metodo `progress` simula il passaggio di un dato numero di secondi, andando ad aggiornare la posizione del corpo. Il metodo `toString` va ridefinito in modo da mostrare la posizione corrente del corpo.

Esempio d'uso:

```
Body b = new Body(0, 0);
b.setSpeed(1,-1.5);
System.out.println(b);
b.progress(1);
System.out.println(b);
b.progress(2);
System.out.println(b);
```

Output del codice precedente:

```
0.0, 0.0
1.0, -1.5
3.0, -4.5
```

232. (20 punti, 26 Giugno 2006) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
public class A {
    public int f() { return 1; }
    public int f(A x) { return f() + 1; }
}
public class B extends A {
    public int f() { return 3; }
    public int f(B x) { return f() + 10; }
}
public class C extends B {
    public int f(C x) { return 5; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        System.out.println(beta.f(beta));
        System.out.println(gamma.f(beta));
        System.out.println(523 < 523);
        System.out.println(257 & 1);
    }
}
```

- Indicare l'output del programma. (15 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (5 punti)

233. **Publication.** (40 punti, 26 Giugno 2006) Nel contesto di un software per biblioteche, si implementi una classe `Publication`, che rappresenta una pubblicazione. Ciascuna pubblicazione ha un titolo e una data di uscita. Implementare le sottoclassi `Book` e `Magazine`. Un libro (book) ha anche un codice ISBN (numero intero di 13 cifre), mentre una rivista (magazine) ha un numero progressivo. Inoltre, una pubblicazione può fare riferimento ad altre pubblicazioni tramite riferimenti bibliografici.

Implementare tutti i metodi necessari a rispettare il seguente caso d'uso.

```
public static void main(String[] x) {
    Publication libro = new Book("The_Art_of_Unix_Programming", new Date(1990, 3, 24),
                                123456);
    Publication rivista = new Magazine("Theoretical_Computer_Science", new Date(1985, 4, 13), 74);
    rivista.addReference(libro);

    for (Publication p : rivista.references())
        System.out.println(p);

    libro.addReference(libro);
}
```

Output del codice precedente:

```
The Art of Unix Programming, ISBN: 123456
```

```
Exception in thread "main" java.lang.RuntimeException
    at Publication.addReference(PublicationTest.java:13)
    at PublicationTest.main(PublicationTest.java:59)
```

234. (20 punti, 26 Giugno 2006) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, vale 0.

Vero Falso

- ☐ ☐ Una interfaccia può estendere una classe.
- ☐ ☐ Una interfaccia può estendere un'altra interfaccia.
- ☐ ☐ Una interfaccia può avere campi.
- ☐ ☐ `Integer` è sottoclasse di `Number`.
- ☐ ☐ Linguaggi I è propedeutico per Linguaggi II.
- ☐ ☐ `RuntimeException` è una eccezione non verificata.
- ☐ ☐ `String` è sottoclasse di `Object`.
- ☐ ☐ Ogni oggetto `Thread` corrisponde ad un thread di esecuzione.
- ☐ ☐ I metodi `equals` e `hashCode` di `Object` sono coerenti tra loro.
- ☐ ☐ “`Double d = 3;`” è una istruzione corretta.

235. **DoubleQueue.** (25 punti, 26 Giugno 2006) Implementare la classe `DoubleQueue`, che rappresenta due code con carico bilanciato. Quando viene aggiunto un nuovo elemento alla `DoubleQueue`, l'elemento viene aggiunto alla coda più scarica, cioè a quella che contiene meno elementi.

```
DoubleQueue<Integer> q = new DoubleQueue<Integer>();
q.add(3);
q.add(5);
q.add(7);

System.out.println("Contenuto_della_prima_coda:");
while (!q.isEmpty1())
    System.out.println(q.remove1());
```

```

System.out.println("Contenuto_della_seconda_coda:");
while (!q.isEmpty2())
    System.out.println(q.remove2());

```

Output del codice precedente:

```

Contenuto della prima coda:
3
7
Contenuto della seconda coda:
5

```

236. (15 punti, 26 Giugno 2006) Individuare e descrivere sinteticamente gli errori nel seguente programma.

```

1  class Test {
2      public static void f(List<? extends Number> l) {
3          l.add(new Integer(3));
4      }
5      public static <T> T myGet(Map<T, ? extends T> m, T x) {
6          return m.get(x);
7      }
8
9      public static void main(String args[]) {
10         f(new LinkedList<Integer>());
11         f(new ArrayList<Boolean>());
12         f(new List<Double>());
13         Object o = myGet(new HashMap<Number, Integer>(), new Integer(7));
14     }
15 }

```

237. **Average.** (30 punti, 26 Giugno 2006) Si implementi una classe **Average** che rappresenti la media aritmetica di un elenco di numeri interi. Ogni oggetto deve possedere un metodo **add** che aggiunge un intero all'elenco, ed un metodo **getAverage** che restituisce la media dei valori immessi fino a quel momento. Il tentativo di chiamare **getAverage** prima che venga inserito alcun valore deve portare ad una eccezione. Esempio di utilizzo:

```

public static void main(String[] x) {
    Average a = new Average();
    double d;

    a.add(10);
    a.add(20);
    d = a.getAverage();
    System.out.println("Media_corrente:_ " + d);

    a.add(60);
    d = a.getAverage();
    System.out.println("Media_corrente:_ " + d);
}

```

Output del codice precedente:

```

Media corrente: 15.0
Media corrente: 30.0

```

Dei 30 punti totali, 10 sono riservati a chi implementa una soluzione che non memorizza tutti gli interi inseriti.

238. (20 punti, 26 Giugno 2006) Dato il seguente programma:

```

public class A {
    private int f() { return 1; }
    public int f(int x) { return f() + 1; }
}
public class B extends A {
    public int f(boolean x) { return 3; }
    public int f(double x) { return f(true) + 1; }
}
public class C extends B {
    public int f(boolean x) { return 5; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new C();
        System.out.println(beta.f(1));
        System.out.println(beta.f(1.0));
        System.out.println(523 & 523);
        System.out.println(257 | 257);
    }
}

```

- Indicare l'output del programma. (15 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (5 punti)

239. **BinaryTreePreIterator.** (40 punti, 26 Giugno 2006) Il seguente frammento di classe definisce un nodo in un albero binario.

```

public class BinaryTreeNode {
    private BinaryTreeNode left, right;
    public BinaryTreeNode getLeft() { return left; }
    public BinaryTreeNode getRight() { return right; }
}

```

Si implementi una classe iteratore `BinaryTreePreIterator` che visiti i nodi dell'albero in preorder (ciascun nodo prima dei suoi figli). Tale classe deve poter essere usata nel seguente modo:

```

public static void main(String[] args) {
    BinaryTreeNode root = ...;
    Iterator i = new BinaryTreePreIterator(root);
    while (i.hasNext()) {
        BinaryTreeNode node = (BinaryTreeNode) i.next();
        ...
    }
}

```

240. (10 punti, 26 Giugno 2006) Individuare e correggere gli errori nel seguente programma.

```

/*
 * Questo programma somma due numeri costanti forniti
 * staticamente da programma e ne stampa il risultato.
 */

public class SommaDueNumeri{
    public void main(String[] args){
        System.out.print("Questo programma somma due numeri."),
        i = 185;
        int j = 1936.27; // tasso di conversione lire in evri :-)
        System.out.print("La somma di " + i + " e " + j + " e': ");
        System.out.println(i+j);
    }
}

```

```

    }
}

```

241. **Polinomio.** (30 punti, 12 gennaio 2007)

Un polinomio è una espressione algebrica del tipo $a_0 + a_1x + \dots + a_nx^n$. Si implementi una classe `Polynomial`, dotata di un costruttore che accetta un array contenente i coefficienti $a_0 \dots a_n$, e dei seguenti metodi: `getDegree` restituisce il grado del polinomio; `times` accetta un polinomio `p` come argomento e restituisce un polinomio che rappresenta il prodotto di `this` e `p`; `toString` produce una stringa simile a quella mostrata nel seguente caso d'uso.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> double a1[] = {1, 2, 3}; double a2[] = {2, 2}; Polynomial p1 = new Polynomial(a1) Polynomial p2 = new Polynomial(a2); Polynomial p3 = p1.times(p2); System.out.println(p1); System.out.println(p2); System.out.println(p3); </pre>	<pre> 1.0 + 2.0 x^1 + 3.0 x^2 2.0 + 2.0 x^1 2.0 + 6.0 x^1 + 10.0 x^2 + 6.0 x^3 </pre>

242. (25 punti, 12 gennaio 2007) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

public class A {
    public String f(A other, int n) { return "A:" + n; }
}
public class B extends A {
    public String f(A other, int n) { return "B1:" + n; }
    public String f(A other, long n) { return "B2:" + n; }
}
public class C extends B {
    public String f(Object other, int n) { return "C:" + n; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;
        System.out.println( alfa.f(beta, 3));
        System.out.println(beta.f( alfa , 3));
        System.out.println(gamma.f(alfa, 3));
        System.out.println( alfa.equals(gamma));
    }
}

```

- Indicare l'output del programma. (20 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (5 punti)

243. (20 punti, 12 gennaio 2007) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti, assenza di risposta 0 punti. Se il totale è negativo, vale 0.

Vero Falso

- ☐ ☐ Una classe può implementare più interfacce.
- ☐ ☐ Un'interfaccia può estendere una classe astratta.

- ☐ ☐ La classe `HashSet<Integer>` è sottoclasse di `HashSet<Number>`.
- ☐ ☐ Un metodo *static* può avere un parametro di tipo.
- ☐ ☐ L'interfaccia `List` estende `LinkedList`.
- ☐ ☐ L'interfaccia `Cloneable` contiene soltanto un metodo.
- ☐ ☐ La classe `Class` è astratta.
- ☐ ☐ La firma di un metodo comprende anche il tipo restituito dal metodo.
- ☐ ☐ Istanziare un oggetto della classe `Thread` provoca l'avvio immediato di un nuovo thread di esecuzione.
- ☐ ☐ Una classe può avere un costruttore privato.

244. **Insieme di polinomi.** (25 punti, 12 gennaio 2007) Con riferimento all'esercizio 241, implementare una classe `PolynomialSet`, che rappresenta un insieme di `Polynomial`. La classe deve offrire almeno i seguenti metodi: `add` accetta un `Polynomial` e lo aggiunge all'insieme; `maxDegree` restituisce il massimo grado dei polinomi dell'insieme; `iterator` restituisce un iteratore sui polinomi dell'insieme. Aggiungere all'insieme un polinomio con gli stessi coefficienti di uno che è già presente non ha alcun effetto sull'insieme.

Dire se nella vostra implementazione è necessario modificare la classe `Polynomial`, e perché.

Dei 25 punti, 7 sono riservati a coloro che forniranno una soluzione in cui `maxDegree` richiede tempo costante (cioè, un tempo indipendente dal numero di polinomi presenti nell'insieme).

245. **Polinomio bis.** (30 punti, 7 febbraio 2007) Si consideri la seguente classe `Pair`.

```
public class Pair<T, U>
{
    public Pair(T first, U second) { this.first = first; this.second = second; }
    public T getFirst() { return first; }
    public U getSecond() { return second; }

    private T first;
    private U second;
}
```

Un *polinomio* è una espressione algebrica del tipo $a_0 + a_1x + \dots + a_nx^n$. Si implementi una classe `Polynomial`, dotata di un costruttore che accetta un array contenente i coefficienti $a_0 \dots a_n$. Deve essere possibile iterare sulle coppie (esponente, coefficiente) in cui il coefficiente è diverso da zero. Cioè, `Polynomial` deve implementare `Iterable<Pair<Integer, Double>>`. Infine, il metodo `toString` deve produrre una stringa simile a quella mostrata nel seguente caso d'uso.

Esempio d'uso:	Output dell'esempio d'uso:
<pre>double a1[] = {1, 2, 0, 3}; double a2[] = {0, 2}; Polynomial p1 = new Polynomial(a1); Polynomial p2 = new Polynomial(a2); System.out.println(p1); System.out.println(p2); for (Pair<Integer, Double> p: p1) System.out.println(p.getFirst() + " : " + p.getSecond());</pre>	<pre>1.0 + 2.0 x^1 + 3.0 x^3 2.0 x^1 0 : 1.0 1 : 2.0 3 : 3.0</pre>

246. **Monomio.** (25 punti, 7 febbraio 2007) Un *monomio* è una espressione algebrica del tipo $a_n \cdot x^n$, cioè è un particolare tipo di polinomio composto da un solo termine. Implementare una classe `Monomial` come sottoclasse di `Polynomial`. La classe `Monomial` deve offrire un costruttore che accetta il grado n e il coefficiente a_n che identificano il monomio.

Ridefinire il metodo `equals` in modo che si possano confrontare liberamente polinomi e monomi, con l'ovvio significato matematico di eguaglianza.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> double a1[] = {1, 2, 3}; double a2[] = {0, 0, 0, 5}; Polynomial p1 = new Polynomial(a1) Polynomial p2 = new Polynomial(a2); Polynomial p3 = new Monomial(3, 5); System.out.println(p2); System.out.println(p3); System.out.println(p3.equals(p1)); System.out.println(p3.equals(p2)); System.out.println(p2.equals(p3)); System.out.println(p2.equals((Object) p3)); </pre>	<pre> 5.0 x^3 5.0 x^3 false true true true </pre>

247. (25 punti, 7 febbraio 2007) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

public class A {
    public String f(A other, int n) { return "A:" + n; }
}
public class B extends A {
    public String f(Object other, int n) { return "B1:" + n; }
    public String f(B other, long n) { return "B2:" + n; }
}
public class C extends B {
    public String f(C other, boolean n) { return "C:" + n; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;

        System.out.println( alfa.f(beta, 3));
        System.out.println(beta.f( alfa , 3));
        System.out.println(gamma.f(gamma, 3));
        System.out.println(gamma.f(gamma, 3L));
    }
}

```

- Indicare l'output del programma. (20 punti)
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (5 punti)
248. (20 punti, 7 febbraio 2007) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti, assenza di risposta 0 punti. Se il totale è negativo, vale 0.

Vero Falso

- ☐ ☐ Una classe astratta può contenere campi.
- ☐ ☐ Una classe può essere privata (`private`).
- ☐ ☐ L'interfaccia `Iterator<Integer>` è sotto-interfaccia di `Iterator<Number>`.
- ☐ ☐ Si può dichiarare un riferimento di tipo `?` (jolly).
- ☐ ☐ Si può dichiarare un riferimento di tipo `List<?>`.
- ☐ ☐ Una classe può avere un costruttore privato.

- ☐ ☐ L'istruzione "String s;" costruisce un oggetto di tipo String.
- ☐ ☐ La classe Method è sotto-classe di Class.
- ☐ ☐ Nella chiamata a.f(), le firme candidate sono ricercate a partire dalla classe dichiarata di a.
- ☐ ☐ sleep è un metodo statico della classe Thread.

249. **Interval.** (33 punti, 29 gennaio 2009) Si implementi la classe `Interval`, che rappresenta un intervallo di numeri reali. Un intervallo può essere chiuso oppure aperto, sia a sinistra che a destra. Il metodo `contains` prende come argomento un numero x e restituisce vero se e solo se x appartiene a questo intervallo. Il metodo `join` restituisce l'unione di due intervalli, senza modificarli, sollevando un'eccezione nel caso in cui questa unione non sia un intervallo. Si implementino anche le classi `Open` e `Closed`, in modo da rispettare il seguente caso d'uso.

Esempio d'uso:	Output dell'esempio d'uso:
<pre>Interval i1 = new Interval.Open(5, 10.5); Interval i2 = new Interval.Closed(7, 20); Interval i3 = i1.join(i2); System.out.println(i1.contains(5)); System.out.println(i1); System.out.println(i2); System.out.println(i3);</pre>	<pre>false (5, 10.5) [7, 20] (5, 20]</pre>

250. (27 punti, 29 gennaio 2009) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
abstract class A {
    public int f(int i, long l1, long l2) { return 1; }
    public int f(int i1, int i2, double d) { return 2; }
}

class B extends A {
    public int f(boolean b, double d, long l) { return 3; }
    public int f(boolean b, int i, double d) { return 4; }
    public int f(int i1, int i2, double d) { return 5; }
}

public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(1, 2, 3));
        System.out.println(alfa.f(1, 2, 3.0));
        System.out.println(beta.f(true, 5, 6));
        System.out.println(beta.f(false, 3.0, 4));
        System.out.println(7 & 5);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (15 punti)
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)
251. **Split.** (20 punti, 29 gennaio 2009) Implementare un metodo statico `split` che prende come argomento tre collezioni A , B e C . Senza modificare A , il metodo inserisce metà dei suoi elementi in B e l'altra metà in C . Porre particolare attenzione alla scelta della firma di `split`, in modo che sia la più generale possibile, ma senza utilizzare parametri di tipo inutili.

252. (20 punti, 29 gennaio 2009) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ La classe `Thread` è astratta.
- ☐ ☐ Il metodo `clone` di `Object` è pubblico.
- ☐ ☐ Il metodo `clone` di `Object` effettua una copia superficiale.
- ☐ ☐ Un campo (o attributo) statico può essere `private`.
- ☐ ☐ Un'eccezione non-verificata non può essere catturata.
- ☐ ☐ `getClass` è un metodo della classe `Object`.
- ☐ ☐ Il pattern `Strategy` si può applicare quando un algoritmo utilizza delle *operazioni primitive*.
- ☐ ☐ Nel framework MVC, le classi view ricevono gli eventi dovuti all'interazione con l'utente.
- ☐ ☐ Nel pattern `Decorator`, l'oggetto decoratore maschera (cioè, fa le veci del) l'oggetto decorato.
- ☐ ☐ L'aggiunta di un `Component AWT` dentro un altro `Component` rappresenta un'istanza del pattern `Composite`.

253. **PostIt.** (25 punti, 8 settembre 2008) Un oggetto di tipo `PostIt` rappresenta un breve messaggio incollato (cioè, collegato) ad un oggetto. Il costruttore permette di specificare il messaggio e l'oggetto al quale incollarlo. Il metodo statico `getMessages` prende come argomento un oggetto e restituisce l'elenco dei `PostIt` collegati a quell'oggetto, sotto forma di una lista, oppure `null` se non c'è nessun `PostIt` collegato.

Esempio d'uso:	Output dell'esempio d'uso:
<pre>Object frigorifero = new Object(); Object libro = new Object(); new PostIt(frigorifero, "comprare_il_latte"); new PostIt(libro, "Bello!!"); new PostIt(libro, "restituire_a_Carlo"); List<PostIt> pl = PostIt.getMessages(libro); for (PostIt p: pl) System.out.println(p);</pre>	<pre>Bello!! restituire a Carlo</pre>

254. (25 punti, 8 settembre 2008) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public int f(int a, int b, float c) { return 1; }
    public int f(int a, double b, int c) { return 2; }
    public int f(double a, float b, long c) { return 3; }
    public int f(double a, int b, double c) { return 4; }
}
```

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

```
        A alfa = new A();
```

```
        System.out.println(alfa.f(1, 2, 3));
        System.out.println(alfa.f(1.0, 2, 3));
        System.out.println(alfa.f(1, 2.0, 3));
```



```

        System.out.println(alfa.f(1.0, 2, 3.0));
        System.out.println(true || (1753/81 < 10235/473));
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (13 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)

255. **RunnableWithArg.** (35 punti, 8 settembre 2008) Si consideri la seguente interfaccia.

```

public interface RunnableWithArg<T> {
    void run(T x);
}

```

Un oggetto `RunnableWithArg` è simile ad un oggetto `Runnable`, tranne per il fatto che il suo metodo `run` accetta un argomento.

Si implementi una classe `RunOnSet` che esegue il metodo `run` di un oggetto `RunnableWithArg` su tutti gli oggetti di un dato insieme, *in parallelo*.

Esempio d'uso:	Un possibile output dell'esempio d'uso:
<pre> Set<Integer> s = new HashSet<Integer>(); s.add(3); s.add(13); s.add(88); RunnableWithArg<Integer> r = new RunnableWithArg<Integer>() { public void run(Integer i) { System.out.println(i/2); } }; Thread t = new RunOnSet<Integer>(r, s); t.start(); </pre>	<pre> 1 6 44 </pre>

256. (20 punti, 8 settembre 2008) Discutere della differenza tra classi astratte ed interfacce. In particolare, illustrare le differenze relativamente ai costruttori, ai campi e ai metodi che possono (o non possono) contenere. Infine, illustrare le linee guida per la scelta dell'una o dell'altra tipologia. (Una pagina al massimo)
257. (20 punti, 8 settembre 2008) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Un `Double` occupa più spazio in memoria di un `double`.
- ☐ ☐ `int` è sottotipo di `long`.
- ☐ ☐ `Runnable` è assegnabile a `Object`.
- ☐ ☐ Una classe anonima può avere un costruttore.
- ☐ ☐ `instanceof` opera sul tipo effettivo del suo primo argomento.
- ☐ ☐ Una classe interna statica non può avere metodi di istanza.
- ☐ ☐ Il pattern `Iterator` prevede un metodo per far ripartire l'iteratore daccapo.
- ☐ ☐ Nel framework MVC, le classi `View` si occupano della presentazione dei dati all'utente.
- ☐ ☐ La classe `Java ActionListener` rappresenta un'applicazione del pattern `Observer`.

- □ L'aggiunta di un tasto (JButton) ad una finestra AWT rappresenta un'applicazione del pattern Decorator.

258. **Anagramma.** (23 punti, 15 gennaio 2009) Si implementi un metodo statico che prende come argomenti due stringhe e restituisce *vero* se sono l'una l'anagramma dell'altra e *falso* altrimenti.

259. (27 punti, 15 gennaio 2009) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
abstract class A {
    public abstract String f(A a, B b);
    public int f(B b, C c) { return 1; }
}
class B extends A {
    public String f(A a, B b) { return "2"; }
    public String f(C c, B b) { return "3"; }
    public int f(C c, Object x) { return 4; }
}
class C extends B {
    public String f(C c1, C c2) { return "5"; }
    public String f(A a, B b) { return "6"; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(alfa, null));
        System.out.println(alfa.f(null, gamma));
        System.out.println(beta.f(gamma, alfa));
        System.out.println(gamma.f(beta, beta));
        System.out.println(1 + "1");
    }
}
```

- Indicare l'output del programma. (15 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)

260. **Volo e Passeggero.** (35 punti, 15 gennaio 2009) Si implementino la classe `Volo` e la classe `Passeggero`. Il costruttore della classe `Volo` prende come argomenti l'istante di partenza e l'istante di arrivo del volo (due numeri interi). Il metodo `add` permette di aggiungere un passeggero a questo volo. Se il passeggero che si tenta di inserire è già presente in un volo che si accavalla con questo, il metodo `add` lancia un'eccezione.

Esempio d'uso:	Output dell'esempio d'uso:
<pre>Volo v1 = new Volo(1000, 2000); Volo v2 = new Volo(1500, 3500); Volo v3 = new Volo(3000, 5000); Passeggero mario = new Passeggero("Mario"); Passeggero luigi = new Passeggero("Luigi"); v1.add(mario); v1.add(luigi); v3.add(mario); // La seguente istruzione provoca l'eccezione v2.add(mario);</pre>	<pre>Exception in thread "main"...</pre>

261. (20 punti, 15 gennaio 2009) La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente, permetta la compilazione della classe A e produca l'output indicato.

Inoltre, rispondere alle seguenti domande:

- (a) Quale design pattern si ritrova nel metodo `Collections.sort`?
 (b) Quale ordinamento sui numeri interi realizza la vostra classe B?

	Output richiesto:
<pre> public class A { public static void main(String[] args) { List<Integer> l = new LinkedList<Integer>(); l.add(3); l.add(70); l.add(23); l.add(50); l.add(5); l.add(20); Collections.sort(l, new B()); for (Integer i: l) System.out.println(i); } } </pre>	<pre> 20 50 70 3 5 23 </pre>

262. (20 punti, 15 gennaio 2009) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Un campo statico viene inizializzato al caricamento della classe.
- ☐ ☐ `Object` è assegnabile a `String`.
- ☐ ☐ `Iterable` è un'interfaccia parametrica.
- ☐ ☐ Una classe astratta può estenderne un'altra.
- ☐ ☐ Si può scrivere `"public interface I<Integer> { }"`.
- ☐ ☐ Un metodo statico può essere astratto.
- ☐ ☐ La scelta del layout di un container `AWT` rappresenta un'istanza del pattern `Template Method`.
- ☐ ☐ `Iterator<T>` estende `Iterable<T>`.
- ☐ ☐ Nel framework `MVC`, ogni oggetto *view* comunica con almeno un oggetto *model*.
- ☐ ☐ Il pattern `Observer` prevede un'interfaccia che sarà implementata da tutti gli osservatori.

263. **Molecola.** (23 punti, 19 giugno 2008) Nell'ambito di un programma di chimica, si implementino le classi `Elemento` e `Molecola`. Un elemento è rappresentato solo dalla sua sigla ("O" per ossigeno, etc.). Una molecola è rappresentata dalla sua formula bruta (" H_2O " per acqua, etc.), cioè dal numero di atomi di ciascun elemento presente.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> Elemento ossigeno = new Elemento("O"); Elemento idrogeno = new Elemento("H"); Molecola acqua = new Molecola(); acqua.add(idrogeno, 1); acqua.add(ossigeno, 1); acqua.add(idrogeno, 1); System.out.println(acqua); </pre>	<pre> H2 O </pre>

264. (27 punti, 19 giugno 2008) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

abstract class A {
    public abstract String f(A a1, A a2);
    public String f(B bb, C cc) { return "A2"; }
}
class B extends A {
    public String f(A a1, A a2) { return "B1"; }
    public String f(C cc, B bb) { return "B2"; }
    private String f(Object x, B bb) { return "B3"; }
}
class C extends B {
    public String f(C c1, C c2) { return "C1"; }
    public String f(A a1, A a2) { return "C2"; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(alfa, alfa));
        System.out.println(alfa.f(beta, beta));
        System.out.println(beta.f(beta, gamma));
        System.out.println(gamma.f(gamma, gamma));
        System.out.println(7 & 3);
    }
}

```

- Indicare l'output del programma. (15 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)

265. **RunnableWithProgress.** (35 punti, 19 giugno 2008) Si consideri la seguente interfaccia.

```

public interface RunnableWithProgress extends Runnable {
    int getProgress();
}

```

Un oggetto `RunnableWithProgress` rappresenta un oggetto `Runnable`, che in più dispone di un metodo che restituisce la percentuale di lavoro completata dal metodo `run` fino a quel momento.

Si implementi una classe `ThreadWithProgress` che esegua un oggetto `RunnableWithProgress` mostrando ad intervalli regolari la percentuale di lavoro svolto fino a quel momento. Precisamente, `ThreadWithProgress` deve stampare a video ogni secondo la percentuale di lavoro aggiornata, a meno che la percentuale non sia la stessa del secondo precedente, nel qual caso la stampa viene saltata.

Esempio d'uso:	Un possibile output dell'esempio d'uso:
<pre> RunnableWithProgress r = new RunnableWithProgress() {...}; Thread t = new ThreadWithProgress(r); t.start(); </pre>	<pre> 5% 12% 25% 70% 90% 100% </pre>

266. (20 punti, 19 giugno 2008) La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A<T> {
    private B<?> myb = new B<Integer>();

    private Integer f(T x) {
        T y = myb.copia(x);
        List<? extends Number> l = B.lista();
        int i = myb.f(2);
        boolean b = myb.f(2.0);
        return myb.g();
    }
}
```

267. (20 punti, 19 giugno 2008) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Si può effettuare l'overriding di un metodo statico.
 - ☐ ☐ `int` è sottotipo di `long`.
 - ☐ ☐ `int` è assegnabile a `long`.
 - ☐ ☐ Una variabile locale può essere `private`.
 - ☐ ☐ Si può scrivere `"public class A<T, Integer> { }"`.
 - ☐ ☐ `getClass` è un metodo della classe `Class`.
 - ☐ ☐ Il pattern `Iterator` prevede un metodo per rimuovere l'ultimo oggetto visitato.
 - ☐ ☐ Nel framework `MVC`, le classi controller ricevono gli eventi dovuti all'interazione con l'utente.
 - ☐ ☐ Nel pattern `Observer`, l'oggetto osservato ha un metodo per registrare un nuovo osservatore.
 - ☐ ☐ La scelta del layout di un container `AWT` rappresenta un'istanza del pattern `Strategy`.
268. **Container.** (35 punti, 19 febbraio 2009) Si implementi la classe `Container`, che rappresenta un contenitore per liquidi di dimensione fissata. Ad un contenitore, inizialmente vuoto, si può aggiungere acqua con il metodo `addWater`, che prende come argomento il numero di litri. Il metodo `getAmount` restituisce la quantità d'acqua presente nel contenitore. Il metodo `connect` prende come argomento un altro contenitore, e lo collega a questo con un tubo. Dopo il collegamento, la quantità d'acqua nei due contenitori (e in tutti quelli ad essi collegati) sarà la stessa.

Esempio d'uso:

```
Container a = new Container(), b = new Container(),
          c = new Container(), d = new Container();

a.addWater(12);
d.addWater(8);
a.connect(b);
System.out.println(a.getAmount()+"_"+b.getAmount()+"_"+
                  c.getAmount()+"_"+d.getAmount());

b.connect(c);
System.out.println(a.getAmount()+"_"+b.getAmount()+"_"+
                  c.getAmount()+"_"+d.getAmount());

c.connect(d);
System.out.println(a.getAmount()+"_"+b.getAmount()+"_"+
                  c.getAmount()+"_"+d.getAmount());
```

Output dell'esempio d'uso:

```
6.0 6.0 0.0 8.0
4.0 4.0 4.0 8.0
5.0 5.0 5.0 5.0
```

269. (27 punti, 19 febbraio 2009) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public int f(double a, long b, long c) { return 1; }
    public int f(double a, int b, double c) { return 2; }
}
class B extends A {
    public int f(int a, double b, long c) { return 3; }
    public int f(int a, int b, double c) { return 4; }
    public int f(double a, int b, double c) { return 5; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(1, 2, 3));
        System.out.println(alfa.f(1, 2, 3.0));
        System.out.println(beta.f(1, 2, 3.0));
        System.out.println(beta.f(1, 21, 3));
        System.out.println(762531 & 762531);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (15 punti)
 - Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)
270. **Interleave.** (20 punti, 19 febbraio 2009) Implementare un metodo statico `interleave` che prende come argomento tre `LinkedList`: A , B e C . Senza modificare A e B , il metodo aggiunge tutti gli elementi di A e di B a C , in modo alternato (prima un elemento di A , poi uno di B , poi un altro elemento di A , e così via). Porre particolare attenzione alla scelta della firma di `interleave`, in modo che sia la più generale possibile, ma senza utilizzare parametri di tipo inutili.
271. (18 punti, 19 febbraio 2009) La seguente classe A fa riferimento ad una classe B . Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A .

```
public class A {
    private Comparator<Double> b = new B(null);

    private <T> A f(T x, T y) {
        return new B(x==y);
    }
}
```

272. (20 punti, 19 febbraio 2009) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Un'interfaccia può essere dichiarata `final`.
- ☐ ☐ Una classe interna può essere dichiarata `final`.
- ☐ ☐ Il metodo `equals` di `Object` effettua un confronto di indirizzi.
- ☐ ☐ `LinkedList<Integer>` è assegnabile a `LinkedList<?>`.
- ☐ ☐ E' possibile lanciare con `throw` qualunque oggetto.

- ☐ ☐ La relazione di “sottotipo” è una relazione di equivalenza tra classi (è riflessiva, simmetrica e transitiva).
- ☐ ☐ Il pattern **Template Method** si può applicare quando un algoritmo utilizza delle *operazioni primitive*.
- ☐ ☐ Nel framework **MVC**, le classi **view** si occupano di presentare il modello all’utente.
- ☐ ☐ Nel pattern **Strategy**, si definisce un’interfaccia che rappresenta un algoritmo.
- ☐ ☐ Il pattern **Iterator** si applica ad un aggregato che non vuole esporre la sua struttura interna.

273. **Triangolo.** (35 punti, 21 aprile 2008) Nell’ambito di un programma di geometria, si implementi la classe **Triangolo**, il cui costruttore accetta le misure dei tre lati. Se tali misure non danno luogo ad un triangolo, il costruttore deve lanciare un’eccezione. Il metodo **getArea** restituisce l’area di questo triangolo. Si implementino anche la classe **Triangolo.Rettangolo**, il cui costruttore accetta le misure dei due cateti, e la classe **Triangolo.Isoscele**, il cui costruttore accetta le misure della base e di uno degli altri lati.

Si ricordi che:

- Tre numeri a , b e c possono essere i lati di un triangolo a patto che $a < b + c$, $b < a + c$ e $c < a + b$.
- L’area di un triangolo di lati a , b e c è data da:

$$\sqrt{p \cdot (p - a) \cdot (p - b) \cdot (p - c)} \quad (\text{formula di Erone})$$

dove p è il semiperimetro.

Esempio d’uso (fuori dalla classe Triangolo):	Output dell’esempio d’uso:
<pre> Triangolo x = new Triangolo(10,20,25); Triangolo y = new Triangolo.Rettangolo(5,8); Triangolo z = new Triangolo.Isoscele(6,5); System.out.println(x.getArea()); System.out.println(y.getArea()); System.out.println(z.getArea()); </pre>	<pre> 94.9918 19.9999 12.0 </pre>

274. (27 punti, 21 aprile 2008) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

abstract class A {
    public abstract String f(Object other, int n);
    public String f(B other, long n) { return "A2:" + n; }
}
class B extends A {
    public String f(Object other, int n) { return "B1:" + n; }
    private String f(C other, long n) { return "B2:" + n; }
}
class C extends B {
    public String f(Object other, long n) { return "C1:" + n; }
    public String f(C other, long n) { return "C2:" + n; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(alfa, 4));
    }
}

```

```

        System.out.println(alfa.f(beta, 4L));
        System.out.println(beta.f(gamma, 4L));
        System.out.println(gamma.f(null, 3L));
        System.out.println(175 & 175);
    }
}

```

- Indicare l'output del programma. (15 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)

275. (23 punti, 21 aprile 2008) Con riferimento all'Esercizio 1, ridefinire il metodo `equals` per i triangoli, in modo da considerare uguali i triangoli che hanno lati uguali. Dire esplicitamente in quale classe (o quali classi) va ridefinito il metodo.

276. **CrazyIterator**. (15 punti, 21 aprile 2008) Individuare l'output del seguente programma. Dire se la classe `CrazyIterator` rispetta il contratto dell'interfaccia `Iterator`. In caso negativo, giustificare la risposta.

```

1  public class CrazyIterator implements Iterator {
2      private int n = 0;
3
4      public Object next() {
5          int j;
6          while (true) {
7              for (j=2; j<=n/2 ;j++) if (n % j == 0) break;
8              if (j > n/2) break;
9              else n++;
10         }
11         return new Integer(n);
12     }
13
14     public boolean hasNext() { n++; return true; }
15     public void    remove() { throw new RuntimeException(); }
16
17     public static void main(String[] args) {
18         Iterator i = new CrazyIterator();
19
20         while (i.hasNext() && (Integer)i.next()<10) {
21             System.out.println(i.next());
22         }
23     }
24 }

```

277. (30 punti, 9 luglio 2008) Data la seguente classe.

```

public class Z {
    private Z other;
    private int val;
    ...
}

```

Si considerino le seguenti specifiche alternative per il metodo `equals`. Due oggetti `x` e `y` di tipo `Z` sono uguali se:

- `x.other` e `y.other` puntano allo stesso oggetto **ed** `x.val` è maggiore o uguale di `y.val`;
- `x.other` e `y.other` puntano allo stesso oggetto **ed** `x.val` e `y.val` sono entrambi pari;
- `x.other` e `y.other` puntano allo stesso oggetto **oppure** `x.val` è uguale a `y.val`;
- `x.other` e `y.other` sono entrambi null **oppure** nessuno dei due è null **ed** `x.other.val` è uguale a `y.other.val`.

- Dire quali specifiche sono valide e perché. (20 punti)
- Implementare la specifica (d). (10 punti)

278. (20 punti, 9 luglio 2008) Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
abstract class A {
    public abstract String f(A a, double x);
    public String f(B b, float x) { return "A2"; }
}
class B extends A {
    public String f(A a, double x) { return "B1"; }
    public String f(C c, double x) { return "B2"; }
    private String f(Object o, double x) { return "B3"; }
}
class C extends B {
    public String f(A a, double x){ return "C1"; }
    public String f(B b, float x) { return "C2"; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(gamma, 5));
        System.out.println(beta.f(alfa, 7));
        System.out.println(gamma.f(gamma, 2.0));
        System.out.println(2 > 1);
    }
}
```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

279. **MutexWithLog.** (35 punti, 9 luglio 2008) Implementare la classe `MutexWithLog` che rappresenta un mutex, con i classici metodi `lock` e `unlock`, che in aggiunta scrive un messaggio a video ogni volta che un thread riesce ad acquisirlo e ogni volta che un thread lo rilascia. Il metodo `unlock` deve lanciare un'eccezione se viene chiamato da un thread diverso da quello che ha acquisito il mutex.

Esempio d'uso:	Un possibile output dell'esempio d'uso:
<pre>final MutexWithLog m = new MutexWithLog(); Thread t = new Thread("Secondo") { public void run() { m.lock(); System.out.println("Due!"); m.unlock(); } }; t.start(); m.lock(); System.out.println("Uno!"); m.unlock();</pre>	<pre>"main" ha acquisito il lock Uno! "main" ha rilasciato il lock "Secondo" ha acquisito il lock Due! "Secondo" ha rilasciato il lock</pre>

280. (20 punti, 9 luglio 2008) La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A<T> {
    private B myb = new B(null);

    private int f(T x) {
        Iterator<?> i = myb.new MyIterator();
        String msg = B.f(x);
        double d = myb.g();
        return myb.g();
    }
}
```

281. (20 punti, 9 luglio 2008) Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- ☐ ☐ Una classe astratta può avere campi istanza.
- ☐ ☐ `LinkedList<String>` è sottotipo di `LinkedList<?>`.
- ☐ ☐ `String` è assegnabile a `Object`.
- ☐ ☐ Un costruttore può chiamarne un altro della stessa classe usando la parola chiave `this`.
- ☐ ☐ Una classe interna può essere `private`.
- ☐ ☐ L'interfaccia `Iterable` contiene un metodo che restituisce un iteratore.
- ☐ ☐ Il pattern `Composite` prevede che un contenitore si possa comportare come un oggetto primitivo.
- ☐ ☐ Nel framework MVC, le classi model si occupano della presentazione dei dati agli utenti.
- ☐ ☐ Nel pattern `Observer`, l'osservatore ha un metodo per agganciarsi ad un oggetto da osservare.
- ☐ ☐ Il passaggio di un parametro `Comparator` al metodo `Collections.sort` rappresenta un'istanza del pattern `Template Method`.

Indice Analitico

281 esercizi

Java Collection Framework (collezioni)

MultiSet, es. 2, pag. 1
Bijection, es. 8, pag. 4
BoundedMap, es. 17, pag. 7
Auditorium, es. 21, pag. 8
Panino, es. 28, pag. 11
Social network, es. 31, pag. 12
Controller, es. 36, pag. 14
Relation, es. 46, pag. 18
MakeMap, es. 60, pag. 23
Intersect, es. 64, pag. 24
SelectKeys, es. 70, pag. 26
PartiallyComparable, es. 74, pag. 27
Contest, es. 91, pag. 35
subMap, es. 102, pag. 38
inverseMap, es. 110, pag. 41
Movie, es. 118, pag. 44
composeMaps, es. 120, pag. 44
MultiBuffer, es. 129, pag. 47
Concat, es. 130, pag. 48
City, es. 138, pag. 50
isSorted, es. 144, pag. 52
Color, es. 146, pag. 53
GetByType, es. 148, pag. 54
CountByType, es. 158, pag. 57
Tutor, es. 161, pag. 58
UML, es. 170, pag. 62
BoolExpr, es. 179, pag. 65
Impianto e Apparecchio, es. 190, pag. 69
FunnyOrder, es. 198, pag. 72
Esercizio 207, pag. 76
Highway, es. 209, pag. 76
Recipe, es. 210, pag. 77
Spartito, es. 223, pag. 82
Esercizio 226, pag. 83
Inventory, es. 227, pag. 84
Publication, es. 233, pag. 87
DoubleQueue, es. 235, pag. 87
Esercizio 236, pag. 88
Insieme di polinomi, es. 244, pag. 91
Polinomio bis, es. 245, pag. 91
PostIt, es. 253, pag. 94
Volo e Passeggero, es. 260, pag. 96
Molecola, es. 263, pag. 97

Container, es. 268, pag. 99

Vero o falso

Esercizio 4, pag. 2
Esercizio 9, pag. 4
Esercizio 14, pag. 6
Esercizio 19, pag. 8
Esercizio 24, pag. 9
Esercizio 33, pag. 13
Esercizio 38, pag. 15
Esercizio 43, pag. 16
Esercizio 48, pag. 18
Esercizio 56, pag. 21
Esercizio 61, pag. 23
Esercizio 66, pag. 24
Esercizio 71, pag. 26
Esercizio 75, pag. 28
Esercizio 80, pag. 30
Esercizio 88, pag. 33
Esercizio 93, pag. 35
Esercizio 98, pag. 37
Esercizio 103, pag. 38
Esercizio 111, pag. 41
Esercizio 116, pag. 43
Esercizio 121, pag. 45
Esercizio 126, pag. 46
Esercizio 131, pag. 48
Esercizio 136, pag. 50
Esercizio 145, pag. 53
Esercizio 150, pag. 54
Esercizio 155, pag. 56
Esercizio 160, pag. 58
Esercizio 165, pag. 60
Esercizio 169, pag. 61
Esercizio 178, pag. 65
Esercizio 183, pag. 67
Esercizio 188, pag. 69
Esercizio 194, pag. 70
Esercizio 199, pag. 72
Esercizio 208, pag. 76
Esercizio 214, pag. 78
Esercizio 218, pag. 80
Esercizio 224, pag. 83
Esercizio 230, pag. 85
Esercizio 234, pag. 87
Esercizio 243, pag. 90
Esercizio 248, pag. 92
Esercizio 252, pag. 94
Esercizio 257, pag. 95
Esercizio 262, pag. 97
Esercizio 267, pag. 99
Esercizio 272, pag. 100
Esercizio 281, pag. 104

Binding dinamico

Esercizio 1, pag. 1
Esercizio 5, pag. 2
Esercizio 10, pag. 4
Esercizio 15, pag. 6
Esercizio 20, pag. 8
Esercizio 25, pag. 10
Esercizio 30, pag. 12
Esercizio 34, pag. 13
Esercizio 39, pag. 15
Esercizio 44, pag. 17
Esercizio 49, pag. 19
Esercizio 52, pag. 20
Esercizio 58, pag. 22
Esercizio 63, pag. 23
Esercizio 68, pag. 25
Esercizio 73, pag. 27
Esercizio 77, pag. 29
Esercizio 82, pag. 31
Esercizio 85, pag. 32
Esercizio 89, pag. 34
Esercizio 94, pag. 35
Esercizio 99, pag. 37
Esercizio 104, pag. 39
Esercizio 107, pag. 40
Esercizio 112, pag. 42
Esercizio 117, pag. 43
Esercizio 122, pag. 45
Esercizio 127, pag. 46
Esercizio 132, pag. 48
Esercizio 137, pag. 50
Esercizio 141, pag. 51
Esercizio 147, pag. 53
Esercizio 152, pag. 55
Esercizio 157, pag. 57
Esercizio 162, pag. 59
Esercizio 167, pag. 61
Esercizio 171, pag. 62
Esercizio 175, pag. 64
Esercizio 180, pag. 66
Esercizio 185, pag. 67
Esercizio 191, pag. 70
Esercizio 196, pag. 71
Esercizio 202, pag. 73
Esercizio 205, pag. 75
Esercizio 211, pag. 77
Esercizio 216, pag. 79
Esercizio 222, pag. 82
Esercizio 229, pag. 85
Esercizio 232, pag. 86
Esercizio 238, pag. 88
Esercizio 242, pag. 90
Esercizio 247, pag. 92
Esercizio 250, pag. 93
Esercizio 254, pag. 94
Esercizio 259, pag. 96
Esercizio 264, pag. 98
Esercizio 269, pag. 100

Esercizio 274, pag. 101
Esercizio 278, pag. 103

Trova l'errore

Esercizio 187, pag. 68
Esercizio 203, pag. 74
Esercizio 207, pag. 76
Esercizio 220, pag. 81
Esercizio 226, pag. 83
Esercizio 236, pag. 88
Esercizio 240, pag. 89

Esercizi elementari

Safe, es. 26, pag. 10
Question e Answer, es. 41, pag. 16
Box, es. 53, pag. 20
Time, es. 62, pag. 23
Segment, es. 67, pag. 25
PrintBytes, es. 79, pag. 30
Crosswords, es. 81, pag. 30
Tetris, es. 84, pag. 32
Playlist, es. 108, pag. 40
Color, es. 146, pag. 53
Wall, es. 166, pag. 60
Circle, es. 172, pag. 63
Esercizio 182, pag. 66
ParkingLot, es. 186, pag. 68
Impianto e Apparecchio, es. 190, pag. 69
Aereo, es. 195, pag. 71
Genealogia, es. 200, pag. 73
Esercizio 203, pag. 74
Rational, es. 204, pag. 74
Esercizio 213, pag. 78
FallingBody, es. 215, pag. 79
TreeType, es. 217, pag. 80
Moto accelerato, es. 221, pag. 81
Moto bidimensionale, es. 231, pag. 86
Average, es. 237, pag. 88
Polinomio, es. 241, pag. 90
Interval, es. 249, pag. 93
Esercizio 256, pag. 95
Anagramma, es. 258, pag. 96
Triangolo, es. 273, pag. 101

Scelta della firma

listIntersection, es. 37, pag. 15
difference, es. 47, pag. 18
reverseList, es. 55, pag. 21
product, es. 97, pag. 36
subMap, es. 102, pag. 38
inverseMap, es. 110, pag. 41
isMax, es. 115, pag. 43
composeMaps, es. 120, pag. 44
agree, es. 125, pag. 46
Concat, es. 130, pag. 48

isMax, es. 135, pag. 49
 isSorted, es. 144, pag. 52

Clonazione di oggetti

Anagrammi, es. 7, pag. 3
 Insieme di lettere, es. 12, pag. 5
 Segment, es. 67, pag. 25
 Esercizio 149, pag. 54
 TreeType, es. 217, pag. 80

Riflessione

GetByType, es. 148, pag. 54
 CountByType, es. 158, pag. 57
 SuperclassIterator, es. 219, pag. 81

Iteratori e ciclo for-each

CrazyIterator, es. 51, pag. 19
 IncreasingSubsequence, es. 153, pag. 55
 MyFor, es. 181, pag. 66
 CommonDividers, es. 184, pag. 67
 Selector, es. 197, pag. 71
 AncestorIterator, es. 201, pag. 73
 SuperclassIterator, es. 219, pag. 81
 TwoSteps, es. 225, pag. 83
 Primes, es. 228, pag. 84
 BinaryTreePreIterator, es. 239, pag. 89
 CrazyIterator, es. 276, pag. 102

Confronto e ordinamento tra oggetti

MaxBox, es. 13, pag. 6
 Point, es. 16, pag. 7
 SetComparator, es. 42, pag. 16
 DataSeries, es. 45, pag. 17
 Pizza, es. 50, pag. 19
 Box, es. 53, pag. 20
 Time, es. 62, pag. 23
 Esercizio 69, pag. 25
 PartiallyComparable, es. 74, pag. 27
 Rebus, es. 83, pag. 31
 EmployeeComparator, es. 90, pag. 34
 Playlist, es. 108, pag. 40
 String comparator, es. 128, pag. 47
 IncreasingSubsequence, es. 153, pag. 55
 Triangle 2, es. 156, pag. 56
 Version, es. 168, pag. 61
 Circle, es. 172, pag. 63
 FunnyOrder, es. 198, pag. 72
 Rational, es. 204, pag. 74
 Sorter, es. 212, pag. 78
 Esercizio 261, pag. 97

Programmazione parametrica (generics)

BoundedMap, es. 17, pag. 7
 Relation, es. 46, pag. 18

MakeMap, es. 60, pag. 23
Intersect, es. 64, pag. 24
SelectKeys, es. 70, pag. 26
Esercizio 78, pag. 29
Esercizio 87, pag. 33
Pair, es. 139, pag. 51
Esercizio 164, pag. 60
Esercizio 177, pag. 65
BoolExpr, es. 179, pag. 65
MyFor, es. 181, pag. 66
CommonDividers, es. 184, pag. 67
ParkingLot, es. 186, pag. 68
Esercizio 187, pag. 68
Selector, es. 197, pag. 71
FunnyOrder, es. 198, pag. 72
Polinomio su un campo generico, es. 206, pag. 75
Sorter, es. 212, pag. 78
Esercizio 226, pag. 83
Inventory, es. 227, pag. 84
Esercizio 236, pag. 88
Polinomio bis, es. 245, pag. 91
Split, es. 251, pag. 93
Interleave, es. 270, pag. 100

Multi-threading

Concurrent filter, es. 3, pag. 2
Mystery thread 2, es. 6, pag. 3
Shared object, es. 11, pag. 5
ThreadRace, es. 18, pag. 7
Shared average, es. 23, pag. 9
Mystery thread, es. 29, pag. 11
SimpleThread, es. 35, pag. 14
TimeToFinish, es. 40, pag. 16
ForgetfulSet, es. 54, pag. 21
VoteBox, es. 57, pag. 22
ExecuteInParallel, es. 65, pag. 24
QueueOfTasks, es. 72, pag. 26
MultiProgressBar, es. 76, pag. 28
atLeastOne, es. 92, pag. 35
Alarm, es. 96, pag. 36
Exchanger, es. 101, pag. 38
PriorityExecutor, es. 109, pag. 41
PostOfficeQueue, es. 114, pag. 42
executeWithDeadline, es. 119, pag. 44
concurrentMax, es. 124, pag. 46
MultiBuffer, es. 129, pag. 47
PeriodicTask, es. 133, pag. 49
processArray, es. 143, pag. 52
Auction, es. 151, pag. 54
Elevator, es. 176, pag. 64
Simulazione di ParkingLot, es. 189, pag. 69
DelayIterator, es. 192, pag. 70
Highway, es. 209, pag. 76
Esercizio 220, pag. 81
RunnableWithArg, es. 255, pag. 95
RunnableWithProgress, es. 265, pag. 98

MutexWithLog, es. 279, pag. 103

Uguaglianza tra oggetti

MultiSet, es. 2, pag. 1
 Anagrammi, es. 7, pag. 3
 Insieme di lettere, es. 12, pag. 5
 Cane, es. 22, pag. 9
 Esercizio 27, pag. 11
 Operai, es. 59, pag. 22
 Shape, es. 105, pag. 39
 Shape equals, es. 106, pag. 40
 PeriodicTask, es. 133, pag. 49
 Esercizio 140, pag. 51
 Monomio, es. 246, pag. 91
 Esercizio 275, pag. 102
 Esercizio 277, pag. 102

Classi interne

Controller, es. 36, pag. 14
 Pizza, es. 50, pag. 19
 Esercizio 149, pag. 54
 Esercizio 154, pag. 56
 Esercizio 159, pag. 58
 Esercizio 173, pag. 63
 Washer, es. 174, pag. 63
 Interval, es. 249, pag. 93
 Triangolo, es. 273, pag. 101

Classe mancante

Esercizio 78, pag. 29
 Esercizio 87, pag. 33
 Esercizio 149, pag. 54
 Esercizio 154, pag. 56
 Esercizio 159, pag. 58
 Esercizio 164, pag. 60
 Esercizio 173, pag. 63
 Esercizio 177, pag. 65
 Esercizio 182, pag. 66
 Esercizio 193, pag. 70
 Esercizio 213, pag. 78
 Esercizio 261, pag. 97
 Esercizio 266, pag. 99
 Esercizio 271, pag. 100
 Esercizio 280, pag. 104

Classi enumerate

Panino, es. 28, pag. 11
 NumberType, es. 32, pag. 13
 Pizza, es. 50, pag. 19
 TetrisPiece, es. 86, pag. 33
 Coin, es. 95, pag. 36
 NutrInfo, es. 100, pag. 37
 Note, es. 113, pag. 42
 Note, es. 123, pag. 45
 Status, es. 134, pag. 49

BloodType, es. 142, pag. 52

Cardinal, es. 163, pag. 59