

# Esercizi di Java Avanzato

Università di Napoli "Federico II"  
Corso di Laurea in Informatica  
Corso di Linguaggi di Programmazione II  
Archivio completo degli esercizi d'esame

Marco Faella

6 febbraio 2022

Quest'opera è offerta con licenza Creative Commons Attribuzione-NonCommerciale 4.0 Internazionale. Per visionare il testo della licenza, visitare <https://creativecommons.org/licenses/by-nc/4.0/legalcode.it>.

This work is licensed under the Creative Commons Attribution-NonCommercial 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Autore: Marco Faella, [marfaella@gmail.com](mailto:marfaella@gmail.com)



# Indice

1	Uguaglianza tra oggetti	1
2	Binding dinamico	9
3	Esercizi elementari	53
4	Java Collection Framework (collezioni)	69
5	Scelta della firma	95
6	Trova l'errore	105
7	Design by contract	109
8	Programmazione parametrica (generics)	111
9	Classe mancante	121
10	Classi interne	127
11	Classi enumerate	133
12	Vero o falso	137
13	Clonazione di oggetti	163
14	Riflessione	167
15	Multi-threading	169
16	Iteratori e ciclo <code>for-each</code>	193
17	Criteri di ordinamento tra oggetti	197
18	Indice Cronologico	209



# 1 Uguaglianza tra oggetti

## 1. (Uguaglianza tra cart, 2020-1-24)

Con riferimento alla classe `Cart` dell'esercizio 1, dire quali delle seguenti sono specifiche valide per il metodo `equals`. In caso negativo, motivare la risposta con un controesempio.

Due istanze di `Cart` sono uguali se:

- entrambi i carrelli contengono almeno un prodotto e il prodotto più caro del primo carrello ha lo stesso prezzo del prodotto più caro del secondo carrello;
- entrambi i carrelli sono vuoti, oppure entrambi contengono almeno un prodotto;
- i due carrelli contengono almeno un prodotto in comune;
- i prezzi totali di ciascun carrello differiscono al più di 10 euro.

## 2. (Student, 2019-7-23)

La classe `Student` è caratterizzata dai seguenti campi:

```
private String nome;  
private long matricola;
```

Dire quali dei seguenti sono criteri di uguaglianza validi per tale classe, giustificando la risposta. Due studenti `a` e `b` sono uguali se:

- Hanno lo stesso nome oppure la stessa matricola.
- Hanno la matricola della stessa parità (entrambe pari o entrambe dispari).
- Hanno lo stesso nome, ma matricole diverse.
- Hanno entrambi i nomi `null`, **oppure** nessuno dei due nomi è `null` e hanno la stessa matricola.

Per ognuno dei criteri validi, implementare il metodo `hashCode` in modo coerente con `equals`.

## 3. (Box e ColoredBox, 2019-6-24)

Considerare le seguenti classi:

```
public class Box {  
    private int x, y;  
    ...  
}
```

```
public class ColoredBox extends Box {  
    private String color;  
    ...  
}
```

Sovrascrivere i metodi `equals` e `hashCode` rispettando le seguenti specifiche:

- Due `Box` sono uguali se hanno le stesse dimensioni `x` e `y`.
- Due `ColoredBox` sono uguali se, in aggiunta, hanno anche lo stesso colore.
- Un `Box` non è mai uguale a un `ColoredBox`.
- `hashCode` deve essere coerente con `equals`.

Precisare in quale/i classe/i è necessario sovrascrivere i metodi per ottenere questo comportamento.

## 4. (Fraction, 2018-7-19)

Implementare la classe `Fraction`, che rappresenta una frazione, e la sottoclasse `ReducedFraction`, che rappresenta una frazione irriducibile.

Due oggetti di questi tipi devono essere uguali per `equals` se rappresentano lo stesso numero razionale, anche se uno è di tipo `Fraction` e l'altro `ReducedFraction`.

Oltre a un costruttore che accetta numeratore e denominatore, le due classi offrono il metodo `times`, che calcola il prodotto, restituendo un nuovo oggetto `Fraction`. Il nuovo oggetto deve essere di tipo effettivo `ReducedFraction` se e soltanto se entrambi gli operandi del prodotto sono di tipo effettivo `ReducedFraction`.

*Suggerimento:* per calcolare il massimo comun divisore tra due interi `a` e `b`, si può usare l'istruzione `BigInteger.valueOf(a).gcd(BigInteger.valueOf(b)).intValue()`.

Esempio d'uso:	Output:
<pre> Fraction a = new Fraction(12,30), b = new ReducedFraction     (12,30),     c = new Fraction(1,4), d = c.times(a); System.out.println(a); System.out.println(b); System.out.println(d); System.out.println(a.equals(b)); System.out.println(c.times(b)); </pre>	<pre> 12/30 2/5 12/120 true 2/20 </pre>

#### 5. (Studente, 2018-3-23)

Implementare la classe `Studente` e le due sottoclassi `Triennale` e `Magistrale`. Uno studente è caratterizzato da nome e matricola. Ciascuna sottoclasse ha un prefisso che viene aggiunto a tutte le sue matricole. Due studenti sono considerati uguali da `equals` se hanno lo stesso nome e la stessa matricola (compreso il prefisso).

L'implementazione deve rispettare il seguente esempio d'uso:

Esempio d'uso:	Output:
<pre> Studente.Triennale.setPrefisso("N86"); Studente.Magistrale.setPrefisso("N97");  Object luca1 = new Studente.Triennale("Luca", "004312"),     luca2 = new Studente.Triennale("Luca", "004312"),     anna1 = new Studente.Triennale("Anna", "004312"),     anna2 = new Studente.Magistrale("Anna", "004312");  System.out.println(luca1.equals(luca2)); System.out.println(anna1.equals(anna2)); System.out.println(anna1); </pre>	<pre> true false Anna N86/004312 </pre>

#### 6. (Room equals, 2017-4-26)

Con riferimento alla classe `Room` dell'esercizio 2, dire quali delle seguenti sono specifiche valide per il metodo `equals`, motivando la risposta.

Due istanze di `Room` sono uguali se:

- a) esiste una data in cui entrambe le camere sono prenotate;
- b) entrambe le camere non hanno alcuna prenotazione;
- c) entrambe le camere non hanno alcuna prenotazione, oppure esiste una persona che ha prenotato entrambe le camere (anche in date diverse);
- d) le camere sono state prenotate negli stessi giorni (anche da persone diverse).

#### 7. (Polygon, 2017-2-23)

La classe `Polygon`, di cui si riporta un frammento, rappresenta un poligono nel piano cartesiano.

```

public class Polygon {
    private static class Vertex {
        double x, y;
    }
    private List<Vertex> vertices;
    ...
}

```

Dire quali delle seguenti sono specifiche valide per l'uguaglianza tra oggetti di tipo `Polygon`, giustificando la risposta.

Due poligoni *a* e *b* sono uguali se:

- a) Hanno lo stesso numero di vertici
- b) Sono entrambi triangoli
- c) Hanno gli stessi vertici, anche se in ordine diverso
- d) Hanno almeno un vertice in comune
- e) Si trovano nello stesso quadrante (con tutti i loro vertici)

Implementare la specifica (d) come metodo `equals` di `Polygon`.

#### 8. (Engine, 2016-4-21)

[CROWDGRADER] Realizzare la classe `Engine`, che rappresenta un motore a combustione, caratterizzato da cilindrata (in  $\text{cm}^3$ ) e potenza (in cavalli). Normalmente, due oggetti `Engine` sono uguali se hanno la stessa cilindrata e la stessa potenza. Il metodo `byVolume` converte questo `Engine` in modo che venga confrontata solo la cilindrata. Analogamente, il metodo `byPower` converte questo `Engine` in modo che venga confrontata solo la potenza.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre>Engine a = new Engine(1200, 69), b = new Engine(1200, 75), c =     new Engine(1400, 75); System.out.println(a); System.out.println(a.equals(b));  Engine aVol = a.byVolume(), bVol = b.byVolume(); System.out.println(aVol); System.out.println(aVol.equals(bVol)); System.out.println(a==aVol);  Engine bPow = b.byPower(), cPow = c.byPower(); System.out.println(bPow); System.out.println(bPow.equals(cPow));</pre>	<pre>(1200.0 cm3, 69.0 CV) false (1200.0 cm3, 69.0 CV) true false (1200.0 cm3, 75.0 CV) true</pre>

#### 9. (Soldier, 2016-3-3)

In un gioco, `Soldier` è una sottoclasse di `Unit`. Gli oggetti `Unit` sono dotati di un campo `health` (intero), mentre gli oggetti `Soldier` hanno anche un campo `name` (stringa).

Dire quali delle seguenti sono specifiche valide per l'uguaglianza tra oggetti di queste due classi, giustificando la risposta. (Quando si dice "un X" si intende "un oggetto di tipo effettivo X")

- a) Due `Unit` o due `Soldier` sono uguali se hanno lo stesso `health`. Uno `Unit` non è mai uguale ad un `Soldier`.
- b) Due `Unit` sono uguali se hanno lo stesso `health`. Due `Soldier` sono uguali se hanno lo stesso `health` e `name`. Uno `Unit` non è mai uguale ad un `Soldier`.
- c) Due `Unit` o due `Soldier` sono uguali se hanno `health` maggiore di zero. Uno `Unit` non è mai uguale ad un `Soldier`.
- d) Due `Unit` sono uguali se hanno lo stesso `health`. Due `Soldier` sono uguali se hanno lo stesso `name`. Uno `Unit` è uguale ad un `Soldier` se hanno lo stesso `health`.

#### 10. (Shape, 2014-4-28)

[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>

#### 11. (Shape equals, 2014-4-28)

Con riferimento all'esercizio 10, 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.

#### 12. (PeriodicTask, 2014-3-5)

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:

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



13. (2013-4-29)

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  $b$ .

14. (Cane, 2013-3-22)

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)

15. (MultiSet, 2013-2-11)

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):
<code>MultiSet&lt;Integer&gt; s1 = new MultiSet&lt;Integer&gt;();</code>	
<code>MultiSet&lt;Integer&gt; s2 = new MultiSet&lt;Integer&gt;();</code>	
<code>s1.add(5); s1.add(7); s1.add(5);</code>	7
<code>s2.add(5); s2.add(5); s2.add(7);</code>	5
<code>for (Integer n: s1)</code>	5
<code>    System.out.println(n);</code>	true
<code>System.out.println(s1.equals(s2));</code>	

16. (Insieme di lettere, 2013-1-22)

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>

## 17. (Anagrammi, 2012-9-3)

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>

## 18. (2012-4-23)

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...

- ...hanno il messaggio segreto di pari lunghezza.
- ...hanno la stessa combinazione oppure lo stesso messaggio segreto.
- ...hanno la combinazione maggiore di zero.
- ...la somma delle due combinazioni è un numero pari.
- ...almeno uno dei due messaggi segreti contiene la parola "maggior-domo".

## 19. (Operai, 2011-2-7)

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`.

- 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.
- 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`.
- Due operai di qualunque tipo sono uguali se hanno lo stesso salario.
- Gli operai semplici sono tutti uguali tra loro. Ciascun operaio specializzato è uguale solo a se stesso.

20. (2008-7-9)

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:

- a) `x.other` e `y.other` puntano allo stesso oggetto **ed** `x.val` è maggiore o uguale di `y.val`;
  - b) `x.other` e `y.other` puntano allo stesso oggetto **ed** `x.val` e `y.val` sono entrambi pari;
  - c) `x.other` e `y.other` puntano allo stesso oggetto **oppure** `x.val` è uguale a `y.val`;
  - d) `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)

21. (2008-4-21)

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.

22. (Monomio, 2007-2-7)

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>



## 2 Binding dinamico

### 1. (2019-7-23)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, B y) { return "A1"; }
    private String f(B x, B y) { return "A2_:" + f(x, y); }
    public String f(A x, Object y) { return "A3"; }
}
class B extends A {
    private String f(B x, B y) { return "B1_:" + f(null, (A)y); }
    public String f(Object x, B y) { return "B2_:" + f(x, (A)y); }
    public String f(A x, B y) { return "B3_:" + f(y, y); }
}
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.getClass() == alfa.getClass());
    }
}
```

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

### 2. (2019-6-24)

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(B x, B y) { return "A2" + f(x, y); }
    public String f(A x, Object y) { return "A3"; }
}
class B extends A {
    public String f(B x, B y) { return "B1_+" + f(null, (A)y); }
    private String f(A x, B y) { return "B2_+" + f(y, y); }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(beta, null));
        System.out.println(beta.f(beta, beta));
        System.out.println(beta.getClass() == alfa.getClass());
    }
}
```

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

## 3. (2019-4-29)

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(A a, A b)      { return "A2"; }
    public String f(B a, C b)      { return "A3"; }
}
class B extends A {
    public String f(Object a, A b) { return "B1_+_ " + f(null, new B()); }
    private String f(A a, B b)    { return "B2"; }
}
class C extends B {
    public String f(Object a, B b) { return "C1"; }
    public String f(A a, B b)      { 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(beta, gamma));
        System.out.println(beta.f(beta, beta));
        System.out.println(gamma.f(alfa, null));
        System.out.println(beta instanceof A);
    }
}
```

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

## 4. (2019-3-19)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

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

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

### 5. (2019-2-15)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, A[] y) { return "A1"; }
    public String f(A x, Object y) { return "A2:" + x.f(new C(), y); }
}
class B extends A {
    public String f(C x, A[] y) { return "B1:" + x.f((A)x, y); }
    public String f(A x, A[] y) { return "B2"; }
    public String f(A x, Object[] y) { return "B3"; }
}
class C extends B {
    public String f(A x, B[] y) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        B[] array = new B[10];
        System.out.println(beta.f(gamma, array));
        System.out.println(gamma.f(beta, null));
        System.out.println(beta.f(array[0], null));
    }
}
```

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

### 6. (2019-1-23)

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, A z) { return "A2"; }
    private 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" + f(x, this, z); }
    private String f(A x, B y, B z) { return "B2"; }
    public String f(B 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, alfa));
        System.out.println(beta.f(beta, beta, beta));
        System.out.println(beta.f(alfa, alfa, null));
    }
}
```

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

## 7. (2018-9-17)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, A y) { return "A1"; }
    private String f(A x, Object y) { return "A2"; }
    protected String f(A x, B y) { return "A3"; }
}
class B extends A {
    public String f(B x, B y) { return "B1_+_ " + f(x, (Object)y); }
    public String f(A x, Object y) { return "B2"; }
}
class C extends B {
    public String f(A x, Object y) { return "C1_+_ " + f(x, (B)y); }
    public String f(Object x, A y) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new C();
        A alfa = beta;
        System.out.println(alfa.f(beta, null));
        System.out.println(beta.f(beta, beta));
        System.out.println(beta.f(alfa, (B)null));
    }
}
```

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

## 8. (2018-7-19)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object a, A b) { return "A1"; }
    public String f(A a, B b) { return "A2"; }
}
class B extends A {
    public String f(B a, B b) { return "B1_+_ " + f(a, (A)b); }
    public String f(A a, B b) { return "B2"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(beta, null));
        System.out.println(beta.f(beta, beta));
        System.out.println(beta.f(alfa, null));
    }
}
```

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

## 9. (2018-6-20)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):



```

class A {
    public String f(Object x, A y, B z) { return "A1"; }
    public String f(A x, C y, C z)      { return "A2"; }
}
class B extends A {
    public String f(Object x, A y, A z) { return "B1_+__" + f(null, new B(), y); }
    private String f(A 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(A x, C y, C z) { 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(beta, gamma, gamma));
        System.out.println(beta.f(beta, beta, beta));
        System.out.println(gamma.f(alfa, null, beta));
    }
}

```

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

10. (2018-5-2)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(Object a, A b) { return "A1"; }
    public String f(A a, C b)      { return "A2"; }
}
class B extends A {
    public String f(Object a, A b) { return "B1_+__" + f(null, new B()); }
    private String f(A a, B b)     { return "B2"; }
}
class C extends B {
    public String f(Object a, B b) { return "C1"; }
    public String f(A a, B b)      { 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(beta, gamma));
        System.out.println(beta.f(beta, beta));
        System.out.println(gamma.f(alfa, null));
        System.out.println(beta instanceof A);
    }
}

```

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

## 11. (2018-3-23)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, A y, B z) { return "A1"; }
    private String f(A x, B y, A z)      { return "A2"; }
}
class B extends A {
    public String f(Object x, A y, B z) { return "B1_+_ " + f(null, z, new B()); }
    private String f(B x, B y, B z)     { return "B2"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = (A) beta;

        System.out.println(alfa.f(alfa, beta, beta));
        System.out.println(alfa.f(beta, alfa, null));
        System.out.println(beta.f(beta, beta, beta));
        System.out.println(alfa instanceof B);
    }
}
```

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

## 12. (2018-10-18)

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(gamma.f(gamma, alfa));
        System.out.println(gamma.f(beta, beta));
        System.out.println(1 + "1");
    }
}
```

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

13. (2017-7-20)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, A y, A z) { return "A1"; }
    private String f(A x, B y, B z) { return "A2"; }
}
class B extends A {
    public String f(Object x, A y, A z) { return "B1_+_" + f(null, new B(), new C()); }
    private String f(B x, B y, C z) { return "B2"; }
}
class C extends B {
    public String f(A x, B y, B z) { return "C1_+_" + f(this, this, z); }
    public String f(B x, B y, B z) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;

        System.out.println(beta.f(beta, beta, gamma));
        System.out.println(gamma.f(beta, null, beta));
        System.out.println(gamma.f(alfa, beta, gamma));
        System.out.println( ( (Object)gamma ).equals( (Object)alfa ) );
    }
}
```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (esclusi println ed equals) indicare la lista delle firme candidate.

14. (2017-6-21)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, A y, A z) { return "A1"; }
    private String f(A x, B y, B z) { return "A2"; }
}
class B extends A {
    public String f(A x, B y, B z) { return "B1_+_" + f(null, new B(), z); }
    private String f(B x, B y, C z) { return "B2"; }
}
class C extends B {
    public String f(A x, B y, B z) { return "C1_+_" + f(this, this, z); }
    public String f(B x, B y, B z) { 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, beta, gamma ));
        System.out.println(gamma.f(beta, beta, beta));
        System.out.println(gamma.f(alfa, beta, null));
        System.out.println( ((Object)beta).equals(alfa) );
    }
}
```

- Indicare l'output del programma.

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

## 15. (2017-4-26)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, A y, A z) { return "A1"; }
    private String f(B x, C y, B z) { return "A2"; }
}
class B extends A {
    public String f(Object x, A y, B z) { return "B1_+_ " + f(null, new B(), z); }
    public String f(A x, B y, C z) { return "B2"; }
}
class C extends B {
    public String f(Object x, A y, B z) { return "C1_+_ " + f(this, this, z); }
    public String f(B x, C y, B z) { 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, beta, gamma));
        System.out.println(gamma.f(beta, beta, beta));
        System.out.println(gamma.f(beta, beta, null));
        System.out.println(128 & 4);
    }
}
```

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

## 16. (2017-3-23)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    String f(A x, int z) { return "A1:" + f(this, z/2.0); }
    public String f(A x, double z) { return "A2"; }
}
class B extends A {
    public String f(A x, float z) { return "B1"; }
    public String f(B x, int z) { return "B2"; }
}
class C extends B {
    public String f(A x, int z) { return "C1"; }
    public String f(B x, int z) { 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, 42));
        System.out.println(beta.f(null, 3.14));
        System.out.println(beta.f(beta, 7));
        System.out.println(16 & 32);
    }
}
```

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

17. (2017-2-23)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    private String f(A x, C z) { return "A1"; }
    public String f(A x, B z) { return "A2:" + x.f(this, (C)x); }
}
class B extends A {
    public String f(A x, C z) { return "B1"; }
    public String f(B x, B z) { return "B2"; }
}
class C extends B {
    public String f(B x, B z) { return "C1"; }
    public String f(B x, C z) { 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, null));
        System.out.println(beta.f(alfa, beta));
        System.out.println(beta.f(null, beta));
        System.out.println(alfa.getClass() == A.class);
    }
}
```

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

18. (2017-10-6)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, Object y, B z) { return "A1"; }
    private String f(A x, B y, B z) { return "A2"; }
}
class B extends A {
    public String f(Object x, A y, A z) { return "B1_+_ " + f(null, new B(), new B()); }
    private String f(B x, B y, B z) { return "B2"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;

        System.out.println(alfa.f(alfa, beta, beta));
        System.out.println(beta.f(beta, alfa, alfa));
        System.out.println(beta.f(null, beta, beta));
        System.out.println(beta.equals( (Object)alfa ));
    }
}
```

- Indicare l'output del programma.

- Per ogni chiamata ad un metodo (esclusi `println` ed `equals`) indicare la lista delle firme candidate.

## 19. (2017-1-25)

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"; }
}
class B extends A {
    public String f(B x, A[] y, B z) { return "B1:" + x.f((A)x, y, z); }
    public String f(A x, B[] y, B z) { return "B2"; }
}
class C extends B {
    public String f(A x, A[] y, C z) { return "C1:" + z.f(new C(), y, z); }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A[] array = new A[10];
        System.out.println(beta.f(gamma, array, gamma));
        System.out.println(gamma.f(array[0], null, beta));
        System.out.println(beta == gamma);
    }
}
```

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

## 20. (2016-9-20)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, A[] y) { return "A1"; }
    public String f(A x, Object y) { return "A2:" + x.f(new C(), y); }
}
class B extends A {
    public String f(B x, A[] y) { return "B1:" + x.f((A)x, y); }
    public String f(A x, B[] y) { return "B2"; }
}
class C extends B {
    public String f(A x, A[] y) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A[] array = new A[10];
        System.out.println(beta.f(gamma, array));
        System.out.println(gamma.f(beta, null));
    }
}
```

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

21. (2016-7-21)

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));
        System.out.println((12 & 3) > 0);
    }
}
```

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

22. (2016-6-22)

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(A[] x, B 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, A 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[0], alfa));
        System.out.println(beta.f(arr[0], arr[1]));
        System.out.println(beta.f(arr, beta));
        System.out.println(5871 & 5871);
    }
}
```

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

## 23. (2016-4-21)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object a, A b) { return "A1"; }
    private String f(B a, C b) { return "A2"; }
}
class B extends A {
    public String f(Object a, A b) { return "B1_+_" + f(null, new B()); }
    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); }
    public String f(B a, C b) { 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(beta, gamma));
        System.out.println(gamma.f(beta, beta));
        System.out.println(gamma.f(beta, null));
        System.out.println(8 & 4);
    }
}
```

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

## 24. (2016-3-3)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
abstract class A {
    public abstract String f(A[] x, Object y);
    public String f(Object[] x, A[] y) { return "A2"; }
}
class B extends A {
    public String f(B[] x, Object y) { return "B1"; }
    public String f(A[] x, Object y) { return "B2"; }
    public String f(B[] x, A[] y) { return "B3"; }
}
class C extends B {
    public String f(B[] x, A[] y) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new C();
        A alfa = new B();
        B[] array = new B[10];
        System.out.println(alfa.f(array, beta));
        System.out.println(beta.f(array, beta));
        System.out.println(beta.f(array, array));
        System.out.println(beta.f(null, array));

        Object betaclass = beta.getClass();
        System.out.println(betaclass instanceof B);
        System.out.println(betaclass instanceof C);
    }
}
```



- 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.

25. (2016-1-27)

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, A y) { return "A2"; }
}
class B extends A {
    public String f(A x, Object y) { return "B1"; }
    private String f(A x, A y) { return "B2"; }
    public String f(B x, B y) { return "B3"; }
}
class C extends B {
    public String f(B x, B y) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new C();
        A alfa = beta;
        System.out.println(beta.f((C)alfa, beta));
        System.out.println(beta.f(beta, null));
        System.out.println(beta.f((Object)beta, alfa));
        System.out.println(alfa.f(beta, 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.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

26. (2015-9-21)

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 "B1"; }
    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(beta.f((C)alfa, alfa));
        System.out.println(beta.f(beta, null));
        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.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

## 27. (2015-7-8)

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.

## 28. (2015-6-24)

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.

29. (2015-2-5)

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.

30. (2015-1-20)

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.

## 31. (2014-9-18)

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.

## 32. (2014-7-3)

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.

### 33. (2014-7-28)

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.

### 34. (2014-4-28)

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.

## 35. (2014-3-5)

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.

## 36. (2014-11-3)

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.

### 37. (2014-11-28)

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.

### 38. (2014-1-31)

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.

## 39. (2013-9-25)

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.

## 40. (2013-7-9)

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.

41. (2013-6-25)

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.

42. (2013-4-29)

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.

## 43. (2013-3-22)

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.

## 44. (2013-2-11)

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.

45. (2013-12-16)

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.

## 46. (2013-1-22)

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.

## 47. (2012-9-3)

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.

## 48. (2012-7-9)

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.

49. (2012-6-18)

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.

## 50. (2012-4-23)

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)

## 51. (2011-3-4)

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)

52. (2011-2-7)

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)

53. (2010-9-14)

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)

54. (2010-7-26)

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)

55. (2010-6-28)

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)

56. (2010-5-3)

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)

57. (2010-2-24)

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)

## 58. (2010-11-30)

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)

## 59. (2010-1-22)

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)

60. (2009-9-1'8)

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)

61. (2009-7-9)

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)

## 62. (2009-6-19)

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)

## 63. (2009-4-23)

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)

64. (2009-2-19)

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)

65. (2009-11-27)

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)

66. (2009-1-29)

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)

67. (2009-1-15)

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)

68. (2008-9-8)

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)

69. (2008-7-9)

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.

## 70. (2008-6-19)

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)

## 71. (2008-4-21)

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)

72. (2008-3-27)

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.

73. (2008-2-25)

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.

## 74. (2008-1-30)

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.

## 75. (2007-9-17)

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.

76. (2007-7-20)

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.

77. (2007-6-29)

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.

## 78. (2007-4-26)

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)

## 79. (2007-2-7)

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)

80. (2007-2-23)

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)

81. (2007-1-12)

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)

82. (2006-9-15)

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)

83. (2006-7-17)

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)

84. (2006-6-26)

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)

85. (2006-4-27)

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)



## 3 Esercizi elementari

### 1. (Exchange, 2022-1-26)

Realizzare la classe `Exchange` che rappresenta una borsa valori. Il metodo `setPrice` imposta il prezzo corrente di un titolo quotato. Il metodo `addLowAlert` fa in modo che un dato runnable venga eseguito la prima volta che il prezzo di un dato titolo raggiunge o scende sotto una data soglia. Il runnable viene eseguito immediatamente se il prezzo di quel titolo è già inferiore o uguale alla soglia. Simmetricamente, il metodo `addHighAlert` offre lo stesso servizio, quando il prezzo raggiunge o sale al di sopra di una data soglia. E' possibile impostare più alert per lo stesso titolo, con soglie diverse o uguali tra loro.

La classe `Exchange` deve essere *thread-safe*.

L'implementazione deve rispettare il seguente esempio d'uso.

---

Esempio d'uso:

```
Exchange borsa = new Exchange();
borsa.setPrice("MaxiCom", 10.56);
borsa.setPrice("MegaCorp", 18.2);
borsa.setPrice("SuperMarkt", 3.91);
borsa.addLowAlert("MegaCorp", 17.5, () -> { System.out.println("Below_the_threshold!"); });
borsa.addHighAlert("MaxiCom", 12, () -> { System.out.println("More_than_12!"); });
borsa.addHighAlert("MaxiCom", 20.5, () -> { System.out.println("More_than_20.5!"); });
borsa.setPrice("MaxiCom", 12.3);
```

---

Output:

More than 12!

---

### 2. (Radio, 2021-9-24)

Realizzare le classi `Radio` e `Channel` che rappresentano una radio e una stazione radiofonica. La classe `Radio` offre un costruttore senza argomenti e i seguenti metodi:

- `addChannel` memorizza e restituisce una nuova stazione, caratterizzata da nome e frequenza. Il tentativo di memorizzare una stazione che ha la stessa frequenza di una stazione già memorizzata deve provocare un'eccezione.
- `nearest` accetta una frequenza e restituisce la stazione con la frequenza più vicina a quella data.

Inoltre, se si itera su un oggetto `Radio` si ottiene la sequenza di stazioni inserite, *in ordine crescente di frequenza*.

Fare in modo che l'unico modo per creare oggetti `Channel` sia tramite il metodo `addChannel`.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre>Radio r = new Radio(); Radio.Channel rai1 = r.addChannel("Rai_Radio_Uno", 89.3); Radio.Channel kk = r.addChannel("Radio_Kiss_Kiss", 101.4); Radio.Channel rmc = r.addChannel("Radio_Monte_Carlo", 96.4); ; for (Radio.Channel c: r) {     System.out.println(c); } System.out.println(r.nearest(98.1));</pre>	<pre>Rai Radio Uno (89.3) Radio Monte Carlo (96.4) Radio Kiss Kiss (101.4)  Radio Monte Carlo (96.4)</pre>

## 3. (GreenPass, 2021-7-26)

Realizzare le classi **Person** e **GreenPass** che rappresentano una persona e una certificazione verde. Una persona è identificata dal suo nome. Il metodo **vaccinate** di **Person** accetta come argomento la data di vaccinazione (un intero che rappresenta un giorno) e restituisce un oggetto **GreenPass**. La classe **GreenPass** offre i seguenti metodi:

- **isValidOn** accetta una data e restituisce vero se questa certificazione verde è valida in quella data.
- **belongsTo** accetta un **Person** e restituisce vero se questa certificazione appartiene a quella persona.

La validità di un **GreenPass** è definita dalle seguenti regole:

- se si tratta della prima dose (prima chiamata a **vaccinate** per questa persona), il **GreenPass** è valido per 180 giorni;
- negli altri casi, il **GreenPass** è valido per 270 giorni.

*Opzionale:* Fare in modo che l'unico modo per creare oggetti **GreenPass** sia tramite il metodo **vaccinate** (qualsiasi altro tentativo deve provocare errore di compilazione o eccezione a runtime).

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre>Person aldo = new Person("Aldo"), barbara = new Person("Barbara"); GreenPass p1 = aldo.vaccinate(10), p2 = aldo.vaccinate(250); System.out.println(p1.isValidOn(20)); System.out.println(p1.isValidOn(200)); System.out.println(p1.belongsTo(barbara));</pre>	<pre>true false false</pre>

## 4. (WiFi, 2021-10-26)

Realizzare le classi **WiFi** e **Network**, che rappresentano un elenco di reti WiFi e una singola rete. La classe **WiFi** offre un costruttore senza argomenti e i seguenti metodi:

- **addNetwork**: memorizza e restituisce una nuova rete, caratterizzata da nome (SSID) e intensità del segnale.<sup>1</sup>
- **strongest**: restituisce la rete con l'intensità più alta (più vicina allo zero).

Inoltre, gli oggetti **WiFi** devono essere iterabili, dando la possibilità di scorrere le reti inserite, *in ordine di intensità decrescente*.

La classe **Network** offre soltanto il metodo **updateStrength**, che aggiorna l'intensità del segnale.

Fare in modo che l'unico modo per creare oggetti **Network** sia tramite il metodo **addNetwork**.

L'implementazione deve rispettare il seguente esempio d'uso.

<sup>1</sup>Misurata in dBm (decibel-milliwatt), l'intensità è un numero negativo ed il segnale è più intenso quanto più il valore è vicino allo zero.

Esempio d'uso:	Output:
<pre> WiFi manager = new WiFi(); WiFi.Network home = manager.addNetwork("Vodafone", -40.5); WiFi.Network hotel = manager.addNetwork("Hotel_Vesuvio", -53.05); WiFi.Network neighbor = manager.addNetwork("Casa_Esposito", -48.95); neighbor.updateStrength(-39.6); WiFi.Network x = manager.strongest(); System.out.println(x); </pre>	Casa Esposito (-39.6 dBm)

## 5. (Cartella, 2017-7-20)

Realizzare la classe `Cartella`, che rappresenta una cartella nella Tombola. Una cartella contiene 15 numeri casuali diversi, compresi tra 1 e 90, disposti in 3 righe di 5 numeri, rispettando la seguente regola:

- una riga non può contenere due numeri della stessa “decina”; ad esempio, una riga può contenere 9 e 11, ma non 11 e 13.

Il metodo `segna` accetta il prossimo numero estratto, e controlla se questa cartella ha ottenuto un premio, restituendo `null`, oppure un valore enumerato che rappresenta uno dei premi della Tombola: `AMBO`, `TERNO`, `QUATERNA`, `CINQUINA`, `TOMBOLA` (implementare anche questa enumerazione).

L'implementazione deve rispettare il seguente esempio d'uso:

Esempio d'uso:	Un output possibile:
<pre> Cartella c = new Cartella(); System.out.println(c.segna(1)); System.out.println(c.segna(2)); System.out.println(c.segna(12)); System.out.println(c.segna(22)); System.out.println(c.segna(82)); </pre>	<pre> null null null AMBO null </pre>

## 6. (Book, 2016-7-21)

Implementare la classe **Book**, che rappresenta un libro diviso in capitoli. Il metodo **addChapter** aggiunge un capitolo in coda al libro, caratterizzato da titolo e contenuto. I capitoli sono automaticamente numerati a partire da 1. Il metodo **getChapterName(i)** restituisce il titolo del capitolo *i*-esimo, mentre il metodo **getChapterContent(i)** ne restituisce il contenuto.

Gli oggetti **Book** devono essere clonabili. Inoltre, la classe deve essere dotata di ordinamento naturale, basato sul numero di capitoli.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre>Book b = new Book(); b.addChapter("Prefazione", "Sono passati pochi anni..."); b.addChapter("Introduzione", "Un calcolatore digitale..."); ; b.addChapter("Sistemi di elaborazione", "Un calcolatore..."); Book bb = b.clone(); System.out.println(bb.getChapterContent(1)); System.out.println(bb.getChapterTitle(2));</pre>	<pre>Sono passati pochi anni... Introduzione</pre>

## 7. (GameLevel, 2016-3-3)

Implementare la classe **GameLevel**, che rappresenta un livello in un gioco 2D, in cui un personaggio si muove su una griglia di caselle. Il costruttore accetta le dimensioni del livello (larghezza e altezza). Il metodo **setWall** accetta le coordinate di una casella e mette un muro in quella casella. Il metodo **areConnected** accetta le coordinate di due caselle e restituisce *vero* se e solo se esiste un percorso tra di loro.

Caso d'uso:
<pre>GameLevel map = new GameLevel(3, 3); System.out.println(map.areConnected(0,0,2,2)); map.setWall(0,1); map.setWall(1,1); System.out.println(map.areConnected(0,0,2,2)); map.setWall(2,1); System.out.println(map.areConnected(0,0,2,2));</pre>
Output:
<pre>true true false</pre>

## 8. (Question e Answer, 2015-7-8)

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:
<pre> 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()); </pre>
Output:
<pre> Canada New Mexico </pre>

#### 9. (Box, 2015-2-5)

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>

#### 10. (Playlist, 2014-7-28)

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>

#### 11. (Safe, 2012-4-23)

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>ResettableSafe s = new ResettableSafe(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>

#### 12. (**PrintBytes**, 2011-3-4)

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"

#### 13. (**Time**, 2010-9-14)

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>

#### 14. (**Tetris**, 2010-7-26)

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>

#### 15. (Crosswords, 2010-5-3)

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>

## 16. (Wall, 2010-2-24)

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>

## 17. (Segment, 2010-11-30)

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>

## 18. (Color, 2010-1-22)

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: <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>	Output dell'esempio d'uso: <pre> red (255, 255, 0) green true </pre>
--	---

#### 19. (Circle, 2009-4-23)

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: <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>	Output dell'esempio d'uso: <pre> true 0 </pre>
--	---

#### 20. (Interval, 2009-1-29)

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: <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>	Output dell'esempio d'uso: <pre> false (5, 10.5) [7, 20] (5, 20] </pre>
--	--

#### 21. (Anagramma, 2009-1-15)

Si implementi un metodo statico che prende come argomenti due stringhe e restituisce *vero* se sono l'una l'anagramma dell'altra e *false* altrimenti.

## 22. (2008-9-8)

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)

## 23. (Triangolo, 2008-4-21)

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>

## 24. (Impianto e Apparecchio, 2008-3-27)

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>

25. (2008-2-25)

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;
}
```

26. (2008-1-30)

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;
}
```

27. (Aereo, 2007-9-17)

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.

28. (ParkingLot, 2007-7-20)

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&lt;Integer&gt; pos1 = p.carIn(); Pair&lt;Integer&gt; 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>

## 29. (Rational, 2007-6-29)

- (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:	Output dell'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>	<pre> 1/6 13/30 2/45 </pre>

## 30. (Genealogia, 2007-4-26)

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> false true </pre>

### 31. (2007-4-26)

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 }

```

### 32. (Polinomio, 2007-1-12)

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>

## 33. (FallingBody, 2006-9-15)

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.

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

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>

## 34. (TreeType, 2006-9-15)

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>

### 35. (Moto accelerato, 2006-7-17)

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>

### 36. (Moto bidimensionale, 2006-6-26)

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
```

### 37. (Average, 2006-4-27)

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.



## 4 Java Collection Framework (collezioni)

### 1. (Product e Cart, 2020-1-24)

Nell'ambito di un sistema di commercio elettronico, implementare le classi **Product** e **Cart** (carrello della spesa). Un prodotto è caratterizzato da descrizione e prezzo. Il tentativo di istanziare due oggetti prodotto con la stessa descrizione deve produrre un'eccezione.

Un carrello può contenere diversi prodotti, compresi eventuali duplicati, ed offre le seguenti funzionalità:

- Aggiungere un prodotto
- Rimuovere un prodotto
- Conoscere il prezzo totale dei prodotti inseriti

L'implementazione deve rispettare il seguente caso d'uso:

Esempio d'uso:	Output:
<pre>Product sedia = new Product("Sedia_elegante", 100); Product tavolo = new Product("Tavolo_di_cristallo", 200); Cart cart = new Cart(); cart.add(sedia); cart.add(tavolo); cart.add(sedia); System.out.println(cart.totalPrice()); cart.remove(sedia); System.out.println(cart.totalPrice()); Product sedia2 = new Product("Sedia_elegante", 150);</pre>	<pre>400 300 Exception in thread "main"...</pre>

### 2. (SortedList, 2019-6-24)

Realizzare la classe **SortedList**, che rappresenta una lista di oggetti dotati di ordinamento naturale. Come una normale lista, una **SortedList** accetta duplicati. Inoltre, è iterabile e i suoi iteratori la percorrono in ordine non decrescente.

Nessun metodo di questa classe può avere una complessità superiore a  $O(n)$ , dove  $n$  è la lunghezza della lista.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre>SortedList&lt;Integer&gt; list = new SortedList&lt;&gt;(); list.add(100); list.add(50); list.add(25); list.add(50);  for (Integer n: list )     System.out.println(n);</pre>	<pre>25 50 50 100</pre>

Suggerimento: si consideri il metodo `void add(int i, T element)` dell'interfaccia `List<T>`, che inserisce un elemento alla posizione  $i$ -esima, spostando tutti gli elementi successivi di una posizione.

### 3. (RotatingList, 2019-4-29)

[CROWDGRADER] Realizzare la classe **RotatingList**, che rappresenta una lista in grado di ruotare, con le seguenti funzionalità:

- Un costruttore senza argomenti che crea una lista vuota.

- Il metodo `add` per inserire un elemento in coda.
- I metodi `rotateLeft` e `rotateRight` che ruotano la lista di una posizione.
- Un overriding di `equals` che consideri uguali due liste se contengono gli stessi elementi, anche in ordine diverso e in molteplicità diversa (ad esempio, la lista `[1, 2, 1]` va considerata uguale a `[2, 1, 2, 2]`).

Nota: le collezioni standard sovrascrivono `toString` in modo da stampare il proprio contenuto.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre>RotatingList&lt;Integer&gt; l = new RotatingList&lt;&gt;(); l.add(1); l.add(2); l.add(3); System.out.println(l); l.rotateLeft(); System.out.println(l); l.add(4); System.out.println(l); l.rotateRight(); System.out.println(l);</pre>	<pre>[1, 2, 3] [2, 3, 1] [2, 3, 1, 4] [4, 2, 3, 1]</pre>

#### 4. (Library, 2019-3-19)

Realizzare per una biblioteca le classi `Library` e `Book`. Un oggetto `Book` è caratterizzato dal suo titolo. La classe `Library` offre le seguenti funzionalità:

- Un costruttore senza argomenti che crea una biblioteca vuota.
- Il metodo `addBook` aggiunge un libro alla biblioteca. Se il libro era già stato aggiunto, restituisce `false`.
- Il metodo `loanBook` prende un libro come argomento e lo dà in prestito, a patto che sia disponibile. Se quel libro è già in prestito, restituisce `false`. Se quel libro non è mai stato inserito nella biblioteca, lancia un'eccezione.
- Il metodo `returnBook` prende un libro come argomento e restituisce quel libro alla biblioteca. Se quel libro non era stato prestato col metodo `loanBook`, il metodo `returnBook` lancia un'eccezione.
- Il metodo `printLoans` stampa la lista dei libri attualmente in prestito, *in ordine temporale* (a partire dal libro in prestito da più tempo).

Inoltre, rispondere alla seguente domanda: nella vostra implementazione, qual è la complessità dei metodi `loanBook` e `returnBook`, rispetto al numero di libri  $n$  inseriti nella biblioteca?

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre>Library lib = new Library(); Book a = new Book("a"), b = new Book("b"), c = new Book("c"); System.out.println(lib.addBook(a)); System.out.println(lib.addBook(b)); System.out.println(lib.addBook(c)); System.out.println(lib.addBook(a)); System.out.println(lib.loanBook(b)); System.out.println(lib.loanBook(a)); lib.printLoans();</pre>	<pre>true true true false true true b a</pre>

#### 5. (Cellphone, 2018-9-17)

Implementare la classe `Cellphone`, che rappresenta un'utenza telefonica dotata di un gestore (stringa) e un numero di telefono (stringa). La classe è in grado di calcolare il costo di una telefonata, in base al gestore di partenza, al gestore di arrivo, e alla durata.

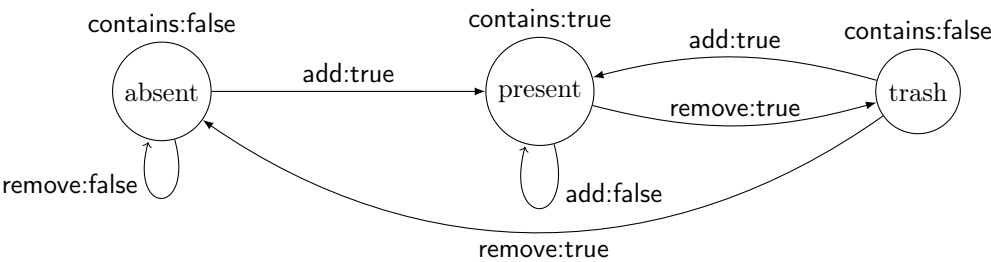
Il metodo `setCost` consente di impostare il costo al minuto di una telefonata con un dato gestore di partenza e un dato gestore di arrivo. Il metodo `getCost` calcola il costo di una telefonata verso una data utenza e di una data durata (in minuti).

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre> Cellphone a = new Cellphone("TIMMY", "3341234"),     b = new Cellphone("Megafon", "3355678"),     c = new Cellphone("Odissey", "3384343"); Cellphone.setCost("TIMMY", "TIMMY", 0.05); Cellphone.setCost("TIMMY", "Megafon", 0.15); Cellphone.setCost("Megafon", "TIMMY", 0.25);  System.out.println(a.getCost(b, 10)); System.out.println(b.getCost(a, 8)); System.out.println(a.getCost(c, 10)); // provoca eccezione </pre>	<pre> 1.5 2.0 Exception in thread "main" ... </pre>

6. (SafeSet, 2018-7-19)

Realizzare la classe `SafeSet`, che rappresenta un insieme che richiede due passaggi per rimuovere completamente un oggetto. Il metodo `add` aggiunge un elemento all'insieme, restituendo `true` se l'inserimento ha avuto successo. Il metodo `remove` rimuove un elemento dall'insieme, ma la rimozione è definitiva solo dopo una seconda chiamata. Il metodo `contains` verifica se l'insieme contiene un dato elemento (in base a `equals`). Infine, un `SafeSet` deve essere *thread-safe*. Il seguente diagramma rappresenta il ciclo di vita di un oggetto all'interno di un `SafeSet`:



L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre> SafeSet&lt;String&gt; a = new SafeSet&lt;&gt;(); System.out.println(a.add("ciao")); System.out.println(a.add("mondo")); System.out.println(a.remove("ciao")); System.out.println(a.contains("ciao")); System.out.println(a.remove("ciao")); System.out.println(a.contains("ciao")); </pre>	<pre> true true true false true false </pre>

7. (Merge, 2018-5-2)

Realizzare un metodo chiamato `merge` che rispetti il seguente contratto:

**Pre-condizione** Accetta due `LinkedList` dello stesso tipo e di pari lunghezza.

**Post-condizione** Restituisce una nuova `LinkedList` ottenuta alternando gli elementi della prima lista e quelli della seconda.

Ad esempio, se la prima lista contiene (1, 2, 3) e la seconda lista (4, 5, 6), la nuova lista deve contenere (1, 4, 2, 5, 3, 6).

**Penale** Se le liste non hanno la stessa lunghezza, lancia `IllegalArgumentException`.

8. (**isSetSmaller**, 2018-3-23)

Implementare il metodo statico `isSetSmaller`, che accetta due insiemi e un comparatore, e restituisce vero se e solo se tutti gli elementi del primo insieme sono più piccoli, in base al comparatore, di tutti gli elementi del secondo insieme.

Porre particolare attenzione alla scelta della firma.

9. (**Book e Library**, 2018-2-22)

Realizzare le classi `Book` e `Library`, che rappresentano rispettivamente un libro e una collezione di libri. Il metodo `addBook` di `Library` aggiunge un libro alla collezione, con un dato titolo e un dato autore. A ciascun libro è possibile attribuire uno o più argomenti tramite il suo metodo `addTag`. Il metodo `getBooksByTag` di `Library` restituisce *in tempo costante* l'insieme dei libri di un argomento dato.

L'implementazione deve rispettare il seguente esempio d'uso:

---

Esempio d'uso:

```
Library casa = new Library(), ufficio = new Library();

Library.Book b1 = casa.addBook("Esercizi_di_stile", "Queneau");
b1.addTag("letteratura");
b1.addTag("umorismo");
Library.Book b2 = casa.addBook("Me_parlare_bene_un_giorno", "Sedaris");
b2.addTag("umorismo");
Library.Book b3 = ufficio.addBook("Literate_programming", "Knuth");
b3.addTag("programmazione");

Set<Library.Book> humorCasa = casa.getBooksByTag("umorismo");
System.out.println(humorCasa);
Set<Library.Book> humorUfficio = ufficio.getBooksByTag("umorismo");
System.out.println(humorUfficio);
```

---

Output:

```
[Esercizi di stile, by Queneau, Me parlare bene un giorno, by Sedaris]
null
```

---

10. (**Component e Configuration**, 2018-10-18)

Un sito vuole consentire agli utenti di ordinare computer assemblati. Data l'enumerazione:

```
enum Type { CPU, BOARD, RAM; }
```

implementare le classi `Component`, che rappresenta un componente di un PC, e `Configuration`, che rappresenta un PC da assemblare.

Un componente è caratterizzato dalla sua tipologia (`Type`) e da una descrizione (stringa). Il suo metodo `setIncompatible` dichiara che questo componente è incompatibile con un altro componente, passato come argomento. Un componente può essere incompatibile con diversi altri componenti.

Il metodo `add` di `Configuration` aggiunge un componente a questo PC e restituisce `true`, ma solo se il componente è compatibile con quelli già inseriti, ed è di tipo diverso da quelli già inseriti, altrimenti non lo inserisce e restituisce `false`.

*Suggerimento:* Una classe `Component` ben progettata non nominerà le 3 istanze di `Type`.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre> Component cpu1 = new Component(Type.CPU, "Ryzen_5_2600"), cpu2 = new Component(Type.CPU, "Core_i5_7500"), board1 = new Component(Type.BOARD, "Prime_X470"), board2 = new Component(Type.BOARD, "Prime_Z370"), ram = new Component(Type.RAM, "DDR4_8GB");  cpu1.setIncompatible(board2); board1.setIncompatible(cpu2);  Configuration pc = new Configuration(); System.out.println(pc.add(cpu1)); System.out.println(pc.add(cpu2)); // due cpu! System.out.println(pc.add(board2)); // incompatibile! System.out.println(pc.add(board1)); System.out.println(pc.add(ram)); </pre>	<pre> true false false true true </pre>

#### 11. (Bug, 2018-1-24)

Realizzare la classe **Bug**, che rappresenta un errore in un programma. Il costruttore accetta una descrizione dell'errore. Inizialmente, l'errore non è assegnato ad alcuno sviluppatore. Il metodo **assignTo** assegna l'errore ad uno sviluppatore, identificato dal nome, che sarà incaricato di risolvere l'errore.

Il metodo statico **getUnassigned** restituisce *in tempo costante* l'insieme degli errori non ancora assegnati. Il metodo statico **getAssignedTo** restituisce *in tempo costante* l'insieme degli errori assegnati ad uno sviluppatore dato.

Nota: un bug assegnato ad uno sviluppatore può essere riassegnato ad un altro.

L'implementazione deve rispettare il seguente esempio d'uso:

Esempio d'uso:	Output:
<pre> Bug b1 = new Bug("Application_crashes_on_Windows_8"), b2 = new Bug("Application_freezes_after_2_hours"), b3 = new Bug("Application_does_not_print_on_laser_ printer"), b4 = new Bug("Data_missing_after_partial_save");  Set&lt;Bug&gt; unassigned = Bug.getUnassigned(); System.out.println(unassigned.size()); b2.assignTo("Paolo"); b3.assignTo("Filomena"); b4.assignTo("Filomena"); System.out.println(unassigned.size()); Set&lt;Bug&gt; filo = Bug.getAssignedTo("Filomena"); System.out.println(filo); </pre>	<pre> 4 1 [("Data missing after partial save", assigned to Filomena), ("Application does not print on laser printer", assigned to Filomena)] </pre>

#### 12. (Room, 2017-4-26)

[CROWDGRADER] Realizzare le classi **Room** e **Reservation**, che rappresentano una camera d'albergo e una prenotazione per la camera. Il metodo **reserve** di **Room** accetta un nome, la data di inizio e di fine prenotazione, e restituisce un oggetto di tipo **Reservation**. Se la camera è occupata in una delle giornate richieste, il metodo lancia un'eccezione. Per semplicità, una data è rappresentata da un numero intero tra 0 a 365. Il metodo **reservations** di **Room** consente di scorrere l'elenco delle prenotazioni, in ordine cronologico.

L'implementazione deve rispettare il seguente esempio d'uso.

<p>Esempio d'uso:</p> <pre>Room r = new Room(); Reservation p1 = r.reserve("Pasquale_Cafiero", 105, 120); Reservation p2 = r.reserve("Carlo_Martello", 5, 20); Reservation p3 = r.reserve("Piero", 20, 22); Reservation p4 = r.reserve("Marinella", 200, 222);  for (Reservation p: r.reservations())     System.out.println(p.getName());</pre>	<p>Output:</p> <pre>Carlo Martello Piero Pasquale Cafiero Marinella</pre>
--	---

## 13. (BinRel, 2017-3-23)

Realizzare la classe `BinRel`, che rappresenta una relazione binaria tra un insieme e se stesso. Il metodo `addPair` aggiunge una coppia di oggetti alla relazione. Il metodo `areRelated` verifica se una data coppia di oggetti appartiene alla relazione. Il metodo `isSymmetric` verifica se la relazione è simmetrica.

L'implementazione deve rispettare il seguente caso d'uso:

<p>Esempio d'uso:</p> <pre>BinRel&lt;String&gt; rel = new BinRel&lt;&gt;(); rel.addPair("a", "albero"); rel.addPair("a", "amaca"); System.out.println(rel.isSymmetric()); rel.addPair("albero", "a"); rel.addPair("amaca", "a"); System.out.println(rel.isSymmetric()); System.out.println(rel.areRelated("a", "amaca"));</pre>	<p>Output:</p> <pre>false true true</pre>
---	---

## 14. (Clinica, 2017-10-6)

Data la seguente enumerazione:

```
enum Specialista { OCULISTA, PEDIATRA; }
```

Realizzare la classe `Clinica`, che permette di prenotare e cancellare visite mediche. I metodi `prenota` e `cancellaPrenotazione` accettano uno specialista e il nome di un paziente, ed effettuano o cancellano la prenotazione, rispettivamente. Il metodo `getPrenotati` restituisce l'elenco dei prenotati.

La classe deve rispettare le seguenti proprietà:

- Non ci si può prenotare con più di uno specialista.
- Si deve poter aggiungere uno specialista all'enumerazione senza dover modificare la classe `Clinica`.

Inoltre, l'implementazione deve rispettare il seguente esempio d'uso:

<p>Esempio d'uso:</p> <pre>Clinica c = new Clinica(); c.prenota(Specialista.OCULISTA, "Pippo_Franco"); c.prenota(Specialista.OCULISTA, "Leo_Gullotta"); c.prenota(Specialista.OCULISTA, "Leo_Gullotta"); c.prenota(Specialista.PEDIATRA, "Ciccio_Ingrassia"); c.prenota(Specialista.PEDIATRA, "Leo_Gullotta"); c.cancellaPrenotazione(Specialista.PEDIATRA, "Ciccio_Ingrassia"); Collection&lt;String&gt; ocu = c.getPrenotati(Specialista.OCULISTA); System.out.println(ocu); System.out.println(c.getPrenotati(Specialista.PEDIATRA));</pre>	<p>Output:</p> <pre>[Leo Gullotta, Pippo Franco] []</pre>
--	---

15. (**mergeIfSorted**, 2017-1-25)

Implementare il metodo statico `mergeIfSorted`, che accetta due liste *a* e *b*, e un comparatore *c*, e restituisce un'altra lista. Inizialmente, usando due thread diversi, il metodo verifica che le liste *a* e *b* siano ordinate in senso non decrescente (ogni thread si occupa di una lista). Poi, se le liste sono effettivamente ordinate, il metodo le fonde (senza modificarle) in un'unica lista ordinata, che viene restituita al chiamante. Se, invece, almeno una delle due liste non è ordinata, il metodo termina restituendo `null`.

Il metodo dovrebbe avere complessità di tempo lineare.

Porre particolare attenzione alla scelta della firma, considerando i criteri di funzionalità, completezza, correttezza, fornitura di garanzie e semplicità.

16. (**SocialUser**, 2016-9-20)

Per un social network, implementare le classi `SocialUser` e `Post`. Un utente è dotato di un nome e può creare dei post tramite il metodo `newPost`. Il contenuto di un post è una stringa, che può contenere nomi di utenti, preceduti dal simbolo "@". Il metodo `getTagged` della classe `Post` restituisce l'insieme degli utenti il cui nome compare in quel post, mentre il metodo `getAuthor` restituisce l'autore del post.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre> SocialUser adriana = new SocialUser("Adriana"),           barbara = new SocialUser("Barbara"); SocialUser.Post p = adriana.newPost("Ecco_una_foto_con_           @Barbara_e_@Carla."); Set&lt;SocialUser&gt; tagged = p.getTagged(); System.out.println(tagged); System.out.println(tagged.iterator().next() == barbara); System.out.println(p.getAuthor()); </pre>	<pre> [Barbara] true Adriana </pre>

Suggerimento: l'invocazione `a.lastIndexOf(b)` restituisce -1 se la stringa *b* non è presente nella stringa *a*, e un numero maggiore o uguale di zero altrimenti.

17. (**GameLevel**, 2016-3-3)

Implementare la classe `GameLevel`, che rappresenta un livello in un gioco 2D, in cui un personaggio si muove su una griglia di caselle. Il costruttore accetta le dimensioni del livello (larghezza e altezza). Il metodo `setWall` accetta le coordinate di una casella e mette un muro in quella casella. Il metodo `areConnected` accetta le coordinate di due caselle e restituisce *vero* se e solo se esiste un percorso tra di loro.

Caso d'uso:	Output:
<pre> GameLevel map = new GameLevel(3, 3); System.out.println(map.areConnected(0,0,2,2)); map.setWall(0,1); map.setWall(1,1); System.out.println(map.areConnected(0,0,2,2)); map.setWall(2,1); System.out.println(map.areConnected(0,0,2,2)); </pre>	<pre> true true false </pre>

18. (**Curriculum**, 2016-1-27)

Un oggetto `Curriculum` rappresenta una sequenza di lavori, ognuno dei quali è un'istanza della classe `Job`. Il costruttore di `Curriculum` accetta il nome di una persona. Il metodo `addJob` aggiunge un lavoro alla sequenza, caratterizzato da una descrizione e dall'anno di inizio, restituendo un

nuovo oggetto di tipo `Job`. Infine, la classe `Job` offre il metodo `next`, che restituisce *in tempo costante* il lavoro successivo nella sequenza (oppure `null`).

Implementare le classi `Curriculum` e `Job`, rispettando il seguente caso d'uso.

---

Caso d'uso:

```
Curriculum cv = new Curriculum("Walter.White");
Curriculum.Job j1 = cv.addJob("Chimico", 1995);
Curriculum.Job j2 = cv.addJob("Insegnante", 2005);
Curriculum.Job j3 = cv.addJob("Cuoco", 2009);

System.out.println(j2.next());
System.out.println(j3.next());
```

---

Output:

```
Cuoco: 2009
null
```

---

#### 19. (Progression, 2015-9-21)

Nell'ambito di un programma di gestione del personale, la classe `Progression` calcola il salario dei dipendenti, in base alla loro anzianità in servizio. Il salario mensile parte da un livello base ed ogni anno solare aumenta di un certo incremento. Il costruttore accetta il salario base e l'incremento annuale. Il metodo `addEmployee` aggiunge un impiegato a questa progressione, specificando il nome e l'anno di assunzione. Il metodo `getSalary` restituisce il salario mensile di un impiegato in un dato anno. Infine, il metodo `addBonus` attribuisce ad un impiegato un bonus extra in un dato anno. Cioè, `addBonus("Pippo", 2010, 50)` significa che Pippo percepirà 50 euro in più in ogni mese del 2010.

---

Caso d'uso:

```
Progression a = new Progression(1000, 150);

a.addEmployee("Jesse", 2008);
a.addEmployee("Gale", 2009);
a.addBonus("Gale", 2010, 300);

System.out.println(a.getSalary("Jesse", 2009));
System.out.println(a.getSalary("Gale", 2010));
System.out.println(a.getSalary("Gale", 2011));
```

---

Output:

```
1150
1450
1300
```

---

#### 20. (Controller, 2015-6-24)

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
```

---

## 21. (Relation, 2015-1-20)

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:

```
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);
```

---

Output:

```
[b, c]
[0, 1]
```

## 22. (Contest, 2014-9-18)

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&lt;String&gt; c = new Contest&lt;String&gt;(); 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

## 23. (subMap, 2014-7-3)

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)`

## 24. (inverseMap, 2014-7-28)

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)`

## 25. (BoundedSet, 2014-1-31)

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&lt;Integer&gt; s = <b>new</b> BoundedSet&lt;Integer&gt;(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>

#### 26. (Movie, 2013-9-25)

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 = <b>new</b> Movie("Rocky", 1976); Movie r2 = <b>new</b> Movie("Rocky_II", 1979, r1); Movie r3 = <b>new</b> Movie("Rocky_III", 1982, r2); Movie f = <b>new</b> Movie("Apocalypse_Now", 1979);  Set&lt;Movie&gt; movies1979 = Movie.selectByYear(1979); System.out.println(movies1979);  List&lt;Movie&gt; rockys = r2.getSeries(); System.out.println(rockys);</pre>	<pre>[Rocky II, Apocalypse Now] [Rocky, Rocky II, Rocky III]</pre>

#### 27. (composeMaps, 2013-9-25)

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.

- $\langle S, T, U \rangle \text{ Map}\langle S, U \rangle \text{ composeMaps}(\text{Map}\langle S, T \rangle a, \text{Map}\langle T, U \rangle b)$
- $\langle S, T, U \rangle \text{ Map}\langle S, U \rangle \text{ composeMaps}(\text{Map}\langle S, T \rangle a, \text{Map}\langle ? \text{ extends } T, U \rangle b)$
- $\langle S, T, U \rangle \text{ Map}\langle S, U \rangle \text{ composeMaps}(\text{Map}\langle S, T \rangle a, \text{Map}\langle ? \text{ super } T, U \rangle b)$
- $\langle S, U \rangle \text{ Map}\langle S, U \rangle \text{ composeMaps}(\text{Map}\langle S, ? \rangle a, \text{Map}\langle ?, U \rangle b)$
- $\langle S, U \rangle \text{ Map}\langle S, U \rangle \text{ composeMaps}(\text{Map}\langle S, \text{Object} \rangle a, \text{Map}\langle \text{Object}, U \rangle b)$

#### 28. (isSorted, 2013-7-9)

Implementare il metodo `isSorted` che accetta una lista e un comparatore, e restituisce `true` se la lista 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.

- a) `boolean isSorted(List<?> x, Comparator<Object> c)`
- b) `<S> boolean isSorted(List<? extends S> x, Comparator<? super S> c)`
- c) `<S> boolean isSorted(List<S> x, Comparator<S> c)`
- d) `boolean isSorted(List<Object> x, Comparator<Object> c)`
- e) `<S, T> boolean isSorted(List<S> x, Comparator<T> c)`
- f) `<S, T extends S> boolean isSorted(List<T> x, Comparator<S> c)`

29. (MultiBuffer, 2013-6-25)

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&lt;Integer&gt; mb = new MultiBuffer&lt;Integer&gt;(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?

30. (Concat, 2013-6-25)

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)`

31. (City, 2013-4-29)

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).

<p>Esempio d'uso:</p> <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&lt;City&gt; r_conn = r.getConnections(); System.out.println(r_conn); System.out.println(r.isConnected(s)); System.out.println(r.isConnected(p));</pre>	<p>Output:</p> <pre>[Napoli] true false</pre>
--	---

### 32. (Auditorium, 2013-3-22)

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.

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

### 33. (MultiSet, 2013-2-11)

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**.

<p>Esempio d'uso:</p> <pre>MultiSet&lt;Integer&gt; s1 = new MultiSet&lt;Integer&gt;(); MultiSet&lt;Integer&gt; s2 = new MultiSet&lt;Integer&gt;(); 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>	<p>Output (l'ordine dei numeri è irrilevante):</p> <pre>7 5 5 true</pre>
--	--

### 34. (Bijection, 2012-9-3)

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:  <pre>Bijection&lt;Integer,String&gt; b = new Bijection&lt;Integer,String&gt;(); 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>	Output:  <pre>dodici null Exception in thread "main" ...</pre>
--	--

### 35. (Social network, 2012-7-9)

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:  <pre>Cleopatra Ottaviano</pre>
---	--

### 36. (BoundedMap, 2012-6-18)

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:  <pre>BoundedMap&lt;String,String&gt; m = new BoundedMap&lt;String,String&gt;(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>	Output dell'esempio d'uso:  <pre>Napoli null</pre>
--	--

### 37. (Panino, 2012-4-23)

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>

38. (**MakeMap, 2011-2-7**)

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.

39. (**Intersect, 2010-9-14**)

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.

40. (**PartiallyComparable, 2010-6-28**)

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&lt;POString&gt; { ...     }  POSet&lt;POString&gt; set = new POSet&lt;POString&gt;(); 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>
--	---

## 41. (SelectKeys, 2010-11-30)

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.

## 42. (Color, 2010-1-22)

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.

<p>Esempio d'uso:</p> <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>	<p>Output dell'esempio d'uso:</p> <pre>red (255, 255, 0) green true</pre>
---	---

## 43. (GetByType, 2010-1-22)

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.

## 44. (Tutor, 2009-6-19)

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 chilometri crescenti, come nel seguente esempio d'uso.

Esempio d'uso:	Output dell'esempio d'uso:
<pre> Tutor tang = <b>new</b> 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>

#### 45. (UML, 2009-4-23)

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 = <b>new</b> UMLClass("Impianto"); UMLClass apparecchio = <b>new</b> UMLClass("Apparecchio"); UMLClass contatore = <b>new</b> UMLClass("Contatore"); <b>new</b> UMLAggregation(apparecchio, impianto, 1, 1); <b>new</b> UMLAggregation(impianto, apparecchio, 0,     UMLAggregation.INFINITY); <b>new</b> 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>

#### 46. (Container, 2009-2-19)

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:	Output dell'esempio d'uso:
<pre> Container a = <b>new</b> Container(), b = <b>new</b> Container(),            c = <b>new</b> Container(), d = <b>new</b> 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()); </pre>	<pre> 6.0  6.0  0.0  8.0 4.0  4.0  4.0  8.0 5.0  5.0  5.0  5.0 </pre>

## 47. (CountByType, 2009-11-27)

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&lt;Number&gt; l = <b>new</b> LinkedList&lt;Number&gt;(); l.add(<b>new</b> Integer(3)); l.add(<b>new</b> Double(4.0)); l.add(<b>new</b> Float(7.0f)); l.add(<b>new</b> Integer(11)); countByType(l); </pre>	<pre> java.lang.Double : 1 java.lang.Float  : 1 java.lang.Integer : 2 </pre>

## 48. (Volo e Passeggero, 2009-1-15)

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 = <b>new</b> Volo(1000, 2000); Volo v2 = <b>new</b> Volo(1500, 3500); Volo v3 = <b>new</b> Volo(3000, 5000); Passeggero mario = <b>new</b> Passeggero("Mario"); Passeggero luigi = <b>new</b> 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>

## 49. (PostIt, 2008-9-8)

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: <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&lt;PostIt&gt; pl = PostIt.getMessage(libro); for (PostIt p: pl)     System.out.println(p); </pre>	Output dell'esempio d'uso: <pre> Bello!! restituire a Carlo </pre>
---	--

#### 50. (Molecola, 2008-6-19)

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: <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>	Output dell'esempio d'uso: <pre> H2 O </pre>
--	--

#### 51. (Impianto e Apparecchio, 2008-3-27)

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: <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>	Output dell'esempio d'uso: <pre> 0 150 180 </pre>
--	---

#### 52. (BoolExpr, 2008-2-25)

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.

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

### 53. (Recipe, 2008-1-30)

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 = <b>new</b> 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>

### 54. (FunnyOrder, 2007-9-17)

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 + "_");
    }
}

```

55. (2007-6-29)

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 }

```

56. (Highway, 2007-6-29)

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>

## 57. (Polinomio bis, 2007-2-7)

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&lt;Integer, Double&gt; 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>

## 58. (Inventory, 2007-2-23)

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&lt;Integer&gt; a = <b>new</b> Inventory&lt;Integer&gt;(); Inventory&lt;String&gt; b = <b>new</b> Inventory&lt;String&gt;();  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>

59. (Insieme di polinomi, 2007-1-12)

Con riferimento all'esercizio 32, 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).

60. (Spartito, 2006-7-17)

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
```

61. (2006-7-17)

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 }

```

## 62. (Publication, 2006-6-26)

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)

```

## 63. (DoubleQueue, 2006-6-26)

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

```

64. (2006-6-26)

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 }

```



## 5 Scelta della firma

### 1. (**combine**, 2022-1-26)

Implementare il metodo `combine`, che accetta due comparatori e li combina *lessicograficamente*, ovvero restituisce un comparatore che, dati due oggetti, restituisce il valore del primo comparatore, se diverso da zero; altrimenti, restituisce il valore del secondo comparatore.

Inoltre, 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) `<T> Comparator<T> combine(Comparator<T> a, Comparator<T> b)`
- b) `<T> Comparator<T> combine(Comparator<T> a, Comparator<?> b)`
- c) `<S, T extends S, U extends S> Comparator<S> combine(Comparator<T> a, Comparator<U> b)`
- d) `<T> Comparator<T> combine(Comparator<? super T> a, Comparator<? super T> b)`
- e) `<T> Comparator<T> combine(Comparator<Object> a, Comparator<Object> b)`
- f) `<T> Comparator<? extends T> combine(Comparator<? super T> a, Comparator<? super T> b)`

### 2. (**countOccurrences**, 2021-9-24)

Implementare il metodo `countOccurrences`, che accetta una collezione e restituisce una mappa che, ad ogni oggetto della collezione, associa il numero di ripetizioni presenti.

Inoltre, 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) `<K> Map<? extends K,Integer> countOccurrences(Collection<Object> c)`
- b) `<K> Map<? extends K,Integer> countOccurrences(Collection<? super K> c)`
- c) `Map<Object,Integer> countOccurrences(Collection<Object> c)`
- d) `<S,T extends S> Map<S,Integer> countOccurrences(Collection<T> c)`
- e) `<K> Map<? super K,Integer> countOccurrences(Collection<? extends K> c)`
- f) `Map<Object,Integer> countOccurrences(Collection<?> c)`

### 3. (**overridingMap**, 2021-7-26)

Il metodo `overridingMap`, accetta due mappe e restituisce una nuova mappa che ha le stesse chiavi della prima e i seguenti valori: se una chiave compare solo nella prima mappa, il suo valore sarà quello associato nella prima mappa; se una chiave compare anche nella seconda mappa, il suo valore sarà quello associato nella seconda mappa.

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) `<K,V> Map<?,?> overridingMap(Map<K,V> map1, Map<K,V> map2)`
- b) `<K> Map<K,?> overridingMap(Map<K,?> map1, Map<K,?> map2)`
- c) `<K,V> Map<K,V> overridingMap(Map<K,V> map1, Map<? extends K,?> map2)`
- d) `<K,V> Map<K,V> overridingMap(Map<K,? extends V> map1, Map<?,? extends V> map2)`

- e) `<K,V> Map<? extends K,V> overridingMap(Map<? super K,V> map1, Map<? super K,V> map2)`
- f) `<K> Map<K,Object> overridingMap(Map<K,Object> map1, Map<?,Object> map2)`

4. (**countInBetween**, 2021-10-26)

Implementare il metodo statico `countInBetween`, che accetta un array, un comparatore e due oggetti *a* e *b*, e restituisce il numero di oggetti dell'array che, secondo il comparatore, sono maggiori di *a* e minori di *b*.

Porre particolare attenzione alla scelta dell'intestazione.

5. (**keysWithHighestValue**, 2020-2-27)

Implementare il metodo `keysWithHighestValue`, che accetta una mappa e un comparatore, e restituisce l'insieme delle chiavi che hanno il massimo valore associato, secondo il comparatore.

Ad esempio, se una `Map<String,Integer>` contiene le coppie `a:5`, `b:3`, `c:0`, `d:5`, e il comparatore rispecchia l'ordine naturale tra interi, il metodo deve restituire l'insieme `{a, d}`.

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) `<K> Set<K> keysWithHighestValue(Map<K,Object> map, Comparator<Object> c)`
- b) `<T> Set<T> keysWithHighestValue(Map<T,T> map, Comparator<T> c)`
- c) `<K,V> Set<K> keysWithHighestValue(Map<K,V> map, Comparator<?> c)`
- d) `<V> Set<Object> keysWithHighestValue(Map<?,? extends V> map, Comparator<? super V> c)`
- e) `<K,V> Set<K> keysWithHighestValue(Map<? super K,? extends V> map, Comparator<? super V> c)`

6. (**disjoin**, 2019-9-20)

Implementare il metodo `disjoin`, che accetta due collezioni e rimuove da entrambe tutti gli oggetti che hanno in comune. Inoltre, restituisce l'insieme degli oggetti rimossi, senza ripetizioni.

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) `<S> Set<S> disjoin(Collection<S> a, Collection<S> b)`
- b) `<S> Set<S> disjoin(Collection<? extends S> a, Collection<?> b)`
- c) `<S,T> Set<? super S> disjoin(Collection<S> a, Collection<T> b)`
- d) `Set<Object> disjoin(List<?> a, List<?> b)`
- e) `<S> Set<S> disjoin(Collection<? super S> a, Collection<? super S> b)`

7. (**Minimum enum**, 2019-7-23)

Implementare il metodo `min`, che accetta due elementi di classi enumerate. Se i due oggetti appartengono alla stessa classe enumerata, il metodo restituisce quello con l'ordinale minore; altrimenti, restituisce `null`.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. In caso di non completezza, presentare un controesempio.

Infine, scegliere l'intestazione migliore oppure proporre un'altra.

- a) `Enum<?> min(Enum<?> a, Enum<?> b)`
- b) `<T> Enum<T> min(Enum<T> a, Enum<T> b)`
- c) `<S extends Enum<S>> S min(S a, S b)`

- d) `<S extends Enum<S>> S min(S a, Object b)`
- e) `<T> Enum<T> min(Enum<T> a, Enum<?> b)`

#### 8. (**keysWithValue**, 2019-6-24)

Il metodo `keysWithValue` accetta una mappa, un valore e una lista, e inserisce nella lista tutte le chiavi della mappa che hanno quel valore associato.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie e semplicità. In caso di non completezza, indicare un controesempio.

Infine, scegliere l'intestazione migliore oppure proporre un'altra.

- a) `<K,V> void keysWithValue(Map<K,V> m, V value, List<K> out)`
- b) `<K> void keysWithValue(Map<K,?> m, Object value, List<Object> out)`
- c) `<K,V> void keysWithValue(Map<? extends K,V> m, V value, List<K> out)`
- d) `<K,V> void keysWithValue(Map<? extends K,V> m, V value, List<? extends K> out)`
- e) `<V> void keysWithValue(Map<?,V> m, V value, List<?> out)`
- f) `<K> void keysWithValue(Map<K,?> m, Object value, LinkedList<? super K> out)`

#### 9. (**interleave2**, 2019-10-9)

Implementare un metodo statico `interleave` che prende come argomento tre liste  $A$ ,  $B$  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).

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie e semplicità.

Infine, scegliere l'intestazione migliore oppure proporre un'altra.

- a) `<S> void interleave(List<S> a, List<S> b, List<S> c)`
- b) `<S, T extends S> void interleave(List<T> a, List<T> b, List<S> c)`
- c) `void interleave(List<?> a, List<?> b, List<Object> c)`
- d) `<S> void interleave(List<?> a, List<?> b, List<S> c)`
- e) `<S> void interleave(List<S> a, List<S> b, List<? super S> c)`

#### 10. (**findPrevious**, 2019-1-23)

Implementare il metodo statico `findPrevious`, che accetta un insieme, un comparatore e un oggetto  $x$ , e restituisce il più grande oggetto dell'insieme che è minore di  $x$  (secondo il comparatore). Se tale oggetto non esiste (perché tutti gli elementi dell'insieme sono maggiori o uguali a  $x$ ), il metodo restituisce `null`.

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) `<T> T findPrevious(Set<? extends T> set, Comparator<?> comp, T x)`
- b) `<S,T extends S> T findPrevious(Set<T> set, Comparator<S> comp, T x)`
- c) `<S,T extends S> S findPrevious(Set<S> set, Comparator<T> comp, S x)`
- d) `<T> T findPrevious(Set<T> set, Comparator<? super T> comp, T x)`
- e) `<T> T findPrevious(Set<T> set, Comparator<T> comp, T x)`
- f) `<T> T findPrevious(Set<? super T> set, Comparator<T> comp, T x)`

#### 11. (**makeMap**, 2018-6-20)

Il metodo statico `makeMap` accetta una lista di chiavi e una lista di valori (di pari lunghezza), e restituisce una mappa ottenuta accoppiando ciascun elemento della prima lista al corrispondente elemento della seconda 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) `<K,V> Map<? extends K,? extends V> makeMap(List<K> keys, List<V> vals)`
- b) `<K,V> Map<? extends K,?> makeMap(List<K> keys, List<?> vals)`
- c) `<K,V> Map<K,V> makeMap(List<K> keys, List<?> vals)`
- d) `<T> Map<T,T> makeMap(List<? extends T> keys, List<? extends T> vals)`
- e) `<K> Map<K,?> makeMap(List<K> keys, List<Object> vals)`
- f) `<K, V extends K> Map<K,V> makeMap(List<K> keys, List<V> vals)`

12. (**cartesianProduct, 2018-2-22**)

Data una classe `Pair<S,T>`, il metodo statico `cartesianProduct` 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) `<S,T> Set<Pair<S,T>> cartesianProduct(Set<S> s, Set<T> t)`
- b) `<S,T> Set<Pair<S,T>> cartesianProduct(Set<?> s, Set<?> t)`
- c) `Set<Pair<?,?>> cartesianProduct(Set<Object> s, Set<Object> t)`
- d) `<S,T> Set<?> cartesianProduct(Set<S> s, Set<T> t)`
- e) `<S> Set<Pair<S,S>> cartesianProduct(Set<S> s, Set<S> t)`
- f) `<S,T extends S> Set<Pair<S,T>> cartesianProduct(Set<S> s, Set<T> t)`

13. (**greatestLowerBound, 2018-10-18**)

Implementare il metodo `gLB` (per *greatestLowerBound*), che accetta due insiemi  $A$  e  $B$ , e un comparatore, e restituisce il più grande elemento di  $A$  che è più piccolo di tutti gli elementi di  $B$ . Se un tale elemento non esiste, il metodo restituisce `null`.

Ad esempio, se  $A = \{5, 20, 30\}$  e  $B = \{25, 26, 30\}$ , il metodo deve restituire 20.

Inoltre, 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) `<T> T gLB(Set<? extends T> a, Set<? extends T> b, Comparator<T> c)`
- b) `<S,T> Object gLB(Set<S> a, Set<T> b, Comparator<Object> c)`
- c) `<T> T gLB(Set<T> a, Set<T> b, Comparator<? super T> c)`
- d) `<S,T extends S> S gLB(Set<S> a, Set<T> b, Comparator<S> c)`
- e) `<T> T gLB(Set<? super T> a, Set<? super T> b, Comparator<T> c)`

14. (**isIncreasing, 2018-1-24**)

Il metodo statico `isIncreasing` accetta una mappa e un comparatore, e restituisce vero se e solo se ciascuna chiave è minore o uguale del valore ad essa associato.

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

- a) `<K,V> boolean isIncreasing(Map<K,V> m, Comparator<K> c)`
- b) `<K,V> boolean isIncreasing(Map<K,V> m, Comparator<? super K> c)`
- c) `<K,V extends K> boolean isIncreasing(Map<K,V> m, Comparator<? super K> c)`
- d) `<T> boolean isIncreasing(Map<T,T> m, Comparator<T> c)`
- e) `<T> boolean isIncreasing(Map<T,T> m, Comparator<? extends T> c)`

- f) `<T> boolean isIncreasing(Map<? extends T, ? extends T> m, Comparator<T> c)`
- g) `boolean isIncreasing(Map<?,?> m, Comparator<?> c)`

15. (**commonKeys, 2017-7-20**)

Implementare un metodo statico `commonKeys`, che accetta due mappe, e restituisce l'insieme degli oggetti che compaiono come chiavi in entrambe le mappe.

Inoltre, 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) `<T> Set<T> commonKeys(Map<T,?> m1, Map<T,?> m2)`
- b) `<T,V1,V2> Set<T> commonKeys(Map<T,V1> m1, Map<T,V2> m2)`
- c) `Set<Object> commonKeys(Map<?,?> m1, Map<?,?> m2)`
- d) `<T> Set<? extends T> commonKeys(Map<? extends T,?> m1, Map<? extends T,?> m2)`
- e) `<T> Set<?> commonKeys(Map<T,?> m1, Map<?,?> m2)`

16. (**findNext, 2017-6-21**)

Il metodo statico `findNext` accetta un insieme, un comparatore e un oggetto `x`, e restituisce il più piccolo oggetto dell'insieme che è maggiore di `x` (secondo il comparatore).

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) `<T> T findNext(Set<? extends T> set, Comparator<?> comp, T x)`
- b) `<S,T extends S> T findNext(Set<T> set, Comparator<S> comp, T x)`
- c) `<S,T extends S> S findNext(Set<S> set, Comparator<T> comp, S x)`
- d) `<T> T findNext(Set<T> set, Comparator<? super T> comp, T x)`
- e) `<T> T findNext(Set<T> set, Comparator<T> comp, Object x)`

17. (**arePermutations, 2016-6-22**)

Il metodo statico `arePermutations`, accetta due liste e controlla se contengono gli stessi elementi (secondo `equals`), anche in ordine diverso.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie e semplicità. Scegliere l'intestazione migliore oppure proporre un'altra. Infine, implementare il metodo usando l'intestazione prescelta.

- a) `boolean arePermutations(List<?> a, List<?> b)`
- b) `<S,T> boolean arePermutations(List<S> a, List<T> b)`
- c) `<S> boolean arePermutations(List<S> a, List<S> b)`
- d) `<S> boolean arePermutations(List<? extends S> a, List<? extends S> b)`
- e) `boolean arePermutations(List<Object> a, List<Object> b)`
- f) `<S, T extends S> boolean arePermutations(List<? extends S> a, List<? extends T> b)`

18. (**splitList, 2015-9-21**)

Il metodo statico `splitList` spezza una lista `src` in due parti, inserendo in una lista `part1` tutti gli elementi che vengono prima di un dato elemento `x`, e tutti gli altri elementi in una lista chiamata `part2`.

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

- a) `<T> void splitList(List<T> src, T x, List<T> part1, List<T> part2)`
- b) `void splitList(List<Object> src, Object x, List<?> part1, List<?> part2)`

- c) `<S,T> void splitList(List<S> src, S x, List<T> part1, List<T> part2)`
- d) `<T> void splitList(List<? extends T> src, T x, List<T> part1, List<T> part2)`
- e) `<T> void splitList(List<T> src, Object x, List<? super T> part1, List<? super T> part2)`

19. (**listIntersection**, 2015-6-24)

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)`

20. (**reverseList**, 2015-2-5)

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)`

21. (**difference**, 2015-1-20)

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)`

22. (**subMap**, 2014-7-3)

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)  $\langle K \rangle \text{ Map} \langle K, ? \rangle \text{ subMap}(\text{Map} \langle K, ? \rangle m, \text{Collection} \langle K \rangle c)$
- b)  $\langle K, V \rangle \text{ Map} \langle K, V \rangle \text{ subMap}(\text{Map} \langle K, V \rangle m, \text{Collection} \langle ? \rangle c)$
- c)  $\langle K, V \rangle \text{ Map} \langle K, V \rangle \text{ subMap}(\text{Map} \langle K, V \rangle m, \text{Collection} \langle ? \text{ super } K \rangle c)$
- d)  $\langle K, V \rangle \text{ Map} \langle K, V \rangle \text{ subMap}(\text{Map} \langle K, V \rangle m, \text{Collection} \langle ? \text{ extends } K \rangle c)$
- e)  $\langle K, V \rangle \text{ Map} \langle K, V \rangle \text{ subMap}(\text{Map} \langle K, V \rangle m, \text{Set} \langle K \rangle c)$
- f)  $\langle K, V, K2 \text{ extends } K \rangle \text{ Map} \langle K, V \rangle \text{ subMap}(\text{Map} \langle K, V \rangle m, \text{Collection} \langle K2 \rangle c)$

23. (**inverseMap**, 2014-7-28)

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 proporla un'altra.

- a)  $\langle K, V \rangle \text{ Map} \langle V, K \rangle \text{ inverseMap}(\text{Map} \langle ?, ? \rangle m)$
- b)  $\text{Map} \langle ?, ? \rangle \text{ inverseMap}(\text{Map} \langle ?, ? \rangle m)$
- c)  $\langle K, V \rangle \text{ Map} \langle K, V \rangle \text{ inverseMap}(\text{Map} \langle V, K \rangle m)$
- d)  $\langle K, V \rangle \text{ Map} \langle K, V \rangle \text{ inverseMap}(\text{Map} \langle ? \text{ extends } V, ? \text{ super } K \rangle m)$
- e)  $\langle K, V \rangle \text{ Map} \langle K, V \rangle \text{ inverseMap}(\text{Map} \langle K, V \rangle m)$
- f)  $\langle K, V \rangle \text{ Map} \langle K, V \rangle \text{ inverseMap}(\text{Map} \langle ? \text{ extends } V, ? \text{ extends } K \rangle m)$

24. (**extractPos**, 2014-3-5)

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 proporla un'altra, motivando brevemente la propria scelta.

- a) `Object extractPos(Collection<?> l, int n)`
- b)  $\langle T \rangle T \text{ extractPos}(\text{List} \langle T \rangle l, \text{int } n)$
- c)  $\langle T \rangle T \text{ extractPos}(\text{List} \langle ? \text{ extends } T \rangle l, \text{int } n)$
- d) `Object extractPos(List<?> l, int n)`
- e)  $\langle T \rangle T \text{ extractPos}(\text{LinkedList} \langle T \rangle l, \text{int } n)$
- f)  $\langle S, T \rangle S \text{ extractPos}(\text{List} \langle T \rangle l, \text{int } n)$

25. (**product**, 2014-11-28)

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 proporla un'altra.

- a)  $\text{Set} \langle \text{Pair} \langle ?, ? \rangle \rangle \text{ product}(\text{Set} \langle ? \rangle a, \text{Set} \langle ? \rangle b)$
- b)  $\langle S, T \rangle \text{ Set} \langle \text{Pair} \langle S, T \rangle \rangle \text{ product}(\text{Set} \langle S \rangle a, \text{Set} \langle T \rangle b)$
- c)  $\text{Set} \langle \text{Pair} \langle \text{Object}, \text{Object} \rangle \rangle \text{ product}(\text{Set} \langle \text{Object} \rangle a, \text{Set} \langle \text{Object} \rangle b)$

26. (**isMax**, 2014-1-31)

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)`

## 27. (**composeMaps**, 2013-9-25)

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)`

## 28. (**isSorted**, 2013-7-9)

Implementare il metodo `isSorted` che accetta una lista e un comparatore, e restituisce `true` se la lista 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.

- a) **boolean** `isSorted(List<?> x, Comparator<Object> c)`
- b) `<S> boolean isSorted(List<? extends S> x, Comparator<? super S> c)`
- c) `<S> boolean isSorted(List<S> x, Comparator<S> c)`
- d) **boolean** `isSorted(List<Object> x, Comparator<Object> c)`
- e) `<S, T> boolean isSorted(List<S> x, Comparator<T> c)`
- f) `<S, T extends S> boolean isSorted(List<T> x, Comparator<S> c)`

## 29. (**Concat**, 2013-6-25)

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)`

30. (**agree, 2013-12-16**)

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.compare(a,b)` e `c2.compare(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)`



## 6 Trova l'errore

### 1. (2007-7-20)

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 }
```

### 2. (2007-6-29)

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 }
```

### 3. (2007-4-26)

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 }

```

## 4. (2006-9-15)

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 }

```

## 5. (2006-7-17)

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 }

```

6. (2006-6-26)

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 }

```

7. (2006-4-27)

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);
    }
}

```





## 7 Design by contract

### 1. (Merge, 2018-5-2)

Realizzare un metodo chiamato `merge` che rispetti il seguente contratto:

**Pre-condizione** Accetta due `LinkedList` dello stesso tipo e di pari lunghezza.

**Post-condizione** Restituisce una nuova `LinkedList` ottenuta alternando gli elementi della prima lista e quelli della seconda.

Ad esempio, se la prima lista contiene (1, 2, 3) e la seconda lista (4, 5, 6), la nuova lista deve contenere (1, 4, 2, 5, 3, 6).

**Penale** Se le liste non hanno la stessa lunghezza, lancia `IllegalArgumentException`.

### 2. (Count, 2016-4-21)

Il metodo `count` accetta una `LinkedList` e restituisce un intero. Il suo contratto è il seguente:

**pre-condizione** La lista contiene stringhe.

**post-condizione** Restituisce la somma delle lunghezze delle stringhe presenti nella lista.

Dire quali dei seguenti sono contratti validi per un overriding di `f`, motivando la risposta.

a) **pre-condizione** Nessuna.

**post-condizione** Restituisce la somma delle lunghezze delle stringhe presenti nella lista (oggetti diversi da stringhe vengono ignorati).

b) **pre-condizione** La lista contiene stringhe non vuote.

**post-condizione** Restituisce la lunghezza della lista.



## 8 Programmazione parametrica (generics)

### 1. (Accumulator, 2020-2-27)

Realizzare la classe parametrica `Accumulator`, che accetta come parametro di tipo una sottoclasse di `Number` e offre i seguenti servizi:

- inserimento di un numero (metodo `add`);
- scorrimento di tutti i numeri inseriti fino a quel momento, divisi tra negativi e non (metodi `negatives` e `positives`);
- somma di tutti i numeri inseriti fino a quel momento (metodo `sum`).

*Suggerimento:* Si ricordi che la classe `Number` prevede il metodo `double doubleValue()`.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre>Accumulator&lt;Integer&gt; acc1 = new Accumulator&lt;&gt;(); acc1.add(10); acc1.add(42); acc1.add(-5); acc1.add(10); for (Integer n: acc1.positives()) System.out.println(n); for (Integer n: acc1.negatives()) System.out.println(n);  Accumulator&lt;Double&gt; acc2 = new Accumulator&lt;&gt;(); acc2.add(-10.0); acc2.add(42.0); System.out.println(acc2.sum());</pre>	<pre>10 42 10 -5 32.0</pre>

Invece, ciascuna delle seguenti due istruzioni deve provocare un errore di compilazione:

```
acc1.positives().add(5);
Accumulator<String> acc3 = new Accumulator<>();
```

### 2. (Range, 2019-2-15)

Realizzare la classe parametrica `Range`, che rappresenta un intervallo di oggetti dotati di ordinamento naturale, con le seguenti funzionalità:

- Il costruttore accetta gli estremi dell'intervallo (l'oggetto minimo e l'oggetto massimo).
- Il metodo `isInside` accetta un oggetto `x` e restituisce `true` se e solo se `x` appartiene all'intervallo.
- Il metodo `isOverlapping` accetta un altro intervallo `x` e restituisce `true` se e solo se `x` è sovrapposto a questo intervallo (cioè se i due hanno intersezione non vuota).
- Il metodo `equals` è ridefinito in modo che due intervalli con gli stessi estremi risultino uguali.
- Il metodo `hashCode` è ridefinito in modo da essere coerente con `equals`.

Porre attenzione alla firma di `isOverlapping` e spiegare se è completa o meno.

L'implementazione deve rispettare il seguente esempio d'uso.

<p>Esempio d'uso:</p> <pre> Range&lt;Integer&gt; a = new Range&lt;&gt;(10, 20); System.out.println(a.isInside(10)); System.out.println(a.isInside(50));  Range&lt;String&gt; b = new Range&lt;&gt;("albero", "dirupo"),     c = new Range&lt;&gt;("casa", "catrame"); System.out.println(b.isOverlapping(c));  Range&lt;Object&gt; d = new Range&lt;&gt;(); // errore di compilazione </pre>	<p>Output:</p> <pre> true false true </pre>
--	---

## 3. (Container, 2017-4-26)

Ipotizzando la disponibilità delle classi `Person`, `Employee` e `Manager`, ciascuna sottoclasse della precedente, realizzare la classe `Container` in modo che il seguente frammento sia corretto:

```

Container<Employee> cont1 = new <String>Container<Employee>("ciao");
Container<Employee> cont2 = new <Integer>Container<Employee>(new Integer(42));
Container<Manager> cont3 = new <Integer>Container<Manager>(new Integer(42));

```

e ciascuna delle seguenti istruzioni provochi un errore di compilazione:

```

Container<Employee> cont4 = new <Object>Container<Employee>(new Object());
Container<Person> cont5 = new <Integer>Container<Person>(new Integer(42));

```

## 4. (2016-7-21)

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<Object> {
    private B<?> b;
    private String msg;
    public A() {
        b = new B<Object>(null);
        msg = B.<A>buildMessage(this);
    }
    public Set<? super Number> f(Set<Integer> set1, Set<String> set2) {
        for (Integer n: b)
            if (b.check(set1, n))
                return b.process(set1, set2, n);
        return b.process(set2, set1, null);
    }
}

```

## 5. (2016-1-27)

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 <S,T extends S> void f(Set<S> set1, Set<T> set2)
    {
        B.process(set1, set2);
        B.process(set2, set1);

        B<S> b = new B<S>();

        S choice1 = b.select(set1),
            choice2 = b.select(set2);
    }
}

```

```

        Collection<? extends S> c = b.filter(set1);
        HashSet<? super S> hs = b.filter(set1);
    }
}

```

#### 6. (Relation, 2015-1-20)

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&lt;Integer,String&gt; r = <b>new</b> Relation&lt;Integer,String&gt;(); 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&lt;String&gt; set0 = r.image(0); Set&lt;Integer&gt; setb = r.preImage("b"); System.out.println(set0); System.out.println(setb); </pre>	<pre> <b>[b, c]</b> <b>[0, 1]</b> </pre>

#### 7. (Pair, 2013-4-29)

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&lt;String,Integer&gt; p1 = <b>new</b> Pair&lt;String,Integer&gt;("uno", 1); System.out.println(p1); </pre>	<pre> <b>(uno, 1)</b> </pre>

#### 8. (BoundedMap, 2012-6-18)

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&lt;String,String&gt; m = <b>new</b> BoundedMap&lt;String,String&gt;(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> <b>Napoli</b> <b>null</b> </pre>

## 9. (2011-3-4)

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;
    }
}
```

## 10. (MakeMap, 2011-2-7)

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.

## 11. (Intersect, 2010-9-14)

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.

## 12. (2010-7-26)

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());
    }
}
```

## 13. (SelectKeys, 2010-11-30)

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.

14. (2009-7-9)

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);
}
```

15. (2009-6-19)

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);
    }
}
```

16. (Interleave, 2009-2-19)

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.

17. (Split, 2009-1-29)

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.

18. (BoolExpr, 2008-2-25)

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.

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

## 19. (MyFor, 2008-2-25)

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) { ... }
```

## 20. (Sorter, 2008-1-30)

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:
<pre> Sorter&lt;Integer&gt; s = <b>new</b> Sorter&lt;Integer&gt;(7);  System.out.println(s.check(4)); System.out.println(s.check(1)); System.out.println(s.check(6)); System.out.println(s.check(6)); </pre>	<pre> -1 -1 1 0 </pre>

## 21. (Selector, 2007-9-17)

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&lt;Integer&gt; l = Arrays.asList(a); Selector&lt;Integer&gt; pari = new Selector&lt;Integer&gt;() {     public boolean select(Integer n) {         return (n % 2) == 0;     } }; for (Integer n: new SelectorIterator&lt;Integer&gt;(l, pari))     System.out.print(n + " ");</pre>	<pre>2 56 22 12 56</pre>

## 22. (FunnyOrder, 2007-9-17)

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 + " ");
    }
}
```

## 23. (CommonDividers, 2007-7-20)

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>

## 24. (ParkingLot, 2007-7-20)

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&lt;Integer&gt; pos1 = p.carIn(); Pair&lt;Integer&gt; 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>

#### 25. (2007-7-20)

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 }
```

#### 26. (Polinomio su un campo generico, 2007-6-29)

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 + 3 x + x^2 Polynomial&lt;Double&gt; p = new Polynomial&lt;Double&gt;(d, new     DoubleField());  System.out.println(p.eval(3.0)); System.out.println(p.eval(2.0));</pre>	<pre>20.0 12.0</pre>

## 27. (Polinomio bis, 2007-2-7)

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&lt;Integer, Double&gt; 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>

## 28. (Inventory, 2007-2-23)

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&lt;Integer&gt; a = <b>new</b> Inventory&lt;Integer&gt;(); Inventory&lt;String&gt; b = <b>new</b> Inventory&lt;String&gt;();  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>

## 29. (2006-7-17)

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 }
```

## 30. (2006-6-26)

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 }
```

## 9 Classe mancante

### 1. (2019-10-9)

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 {  
    Comparator<Double> b = new B(null);  
  
    Comparator<String> c = (x, y) -> B.g(x, y);  
  
    <T> A f(T x, T y) {  
        return new B(x==y);  
    }  
}
```

### 2. (2016-7-21)

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<Object> {  
    private B<?> b;  
    private String msg;  
    public A() {  
        b = new B<Object>(null);  
        msg = B.<A>buildMessage(this);  
    }  
    public Set<? super Number> f(Set<Integer> set1, Set<String> set2) {  
        for (Integer n: b)  
            if (b.check(set1, n))  
                return b.process(set1, set2, n);  
        return b.process(set2, set1, null);  
    }  
}
```

### 3. (2016-1-27)

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 <S,T extends S> void f(Set<S> set1, Set<T> set2)  
    {  
        B.process(set1, set2);  
        B.process(set2, set1);  
  
        B<S> b = new B<S>();  
  
        S choice1 = b.select(set1),  
          choice2 = b.select(set2);  
  
        Collection<? extends S> c = b.filter(set1);  
        HashSet<? super S> hs = b.filter(set1);  
    }  
}
```

## 4. (2011-3-4)

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;
    }
}
```

## 5. (2010-7-26)

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());
    }
}
```

## 6. (2010-1-22)

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;
}
```

7. (2009-9-1'8)

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) { }
}
```

8. (2009-7-9)

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);
}
```

9. (2009-6-19)

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);
    }
}
```

10. (2009-4-23)

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();
    }
}

```

## 11. (2009-2-19)

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);
    }
}

```

## 12. (2009-11-27)

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) { }
}

```

## 13. (2009-1-15)

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&lt;Integer&gt; l = new LinkedList&lt;Integer&gt;();         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>

14. (2008-7-9)

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();
    }
}

```

15. (2008-6-19)

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();
    }
}

```

16. (2008-3-27)

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);
    }
}

```

## 17. (2008-2-25)

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;
}
```

## 18. (2008-1-30)

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;
}
```

## 10 Classi interne

### 1. (InternalLayout2, 2021-9-24)

Date le seguenti classi:

```
class A {  
    public int n = 0;  
    private C c = new C();  
    public static class B {  
        int i = 1;  
    }  
    public class C {  
        int j = 2;  
        public C() {  
            A.this.n++;  
        }  
    }  
}
```

Disegnare il *memory layout* alla fine dell'esecuzione del seguente frammento di codice:

```
A a = new A();  
A.B b = new A.B();  
A.C c = a.new C();
```

### 2. (InternalLayout1, 2021-7-26)

Date le seguenti classi:

```
public class A {  
    public int n;  
    private A myself = this;  
    public static class B extends A {  
        int i = 1;  
    }  
    public class C {  
        int j = 2;  
    }  
}
```

Disegnare il *memory layout* del seguente frammento di codice:

```
A a1 = new A();  
A a2 = new A.B();  
A.C c1 = a1.new C();  
A.C c2 = a2.new C();
```

### 3. (Microwave, 2019-9-20)

L'interfaccia `Microwave` rappresenta i controlli di un forno a microonde, con un metodo per accenderlo impostando la potenza (in watt) e un metodo per spegnerlo.

```
interface Microwave {  
    void on(int power);  
    void off();  
}
```

Implementare la classe `Program`, che rappresenta una sequenza di istruzioni per un forno a microonde, rispettando il seguente caso d'uso.

La seguente riga crea un programma che cuoce per 10 secondi a 500 watt e poi cuoce per altri 5 secondi a 700 watt.

```
Program p = Program.make().on(500).delay(10000).on(700).delay(5000).off();
```

Dato un oggetto `oven` di tipo `Microwave`, la seguente riga esegue il programma `p` sul forno in questione, invocando i metodi `on` e `off` dell'oggetto `oven` al momento giusto.

```
p.executeOn(oven);
```

Il metodo `executeOn` è bloccante e restituisce il controllo al chiamante solo quando il programma è terminato.

#### 4. (Studente, 2018-3-23)

Implementare la classe `Studente` e le due sottoclassi `Triennale` e `Magistrale`. Uno studente è caratterizzato da nome e matricola. Ciascuna sottoclasse ha un prefisso che viene aggiunto a tutte le sue matricole. Due studenti sono considerati uguali da `equals` se hanno lo stesso nome e la stessa matricola (compreso il prefisso).

L'implementazione deve rispettare il seguente esempio d'uso:

Esempio d'uso:	Output:
<pre>Studente.Triennale.setPrefisso("N86"); Studente.Magistrale.setPrefisso("N97");  Object luca1 = new Studente.Triennale("Luca", "004312"),       luca2 = new Studente.Triennale("Luca", "004312"),       anna1 = new Studente.Triennale("Anna", "004312"),       anna2 = new Studente.Magistrale("Anna", "004312");  System.out.println(luca1.equals(luca2)); System.out.println(anna1.equals(anna2)); System.out.println(anna1);</pre>	<pre>true false Anna N86/004312</pre>

#### 5. (Engine, 2016-4-21)

[CROWDGRADER] Realizzare la classe `Engine`, che rappresenta un motore a combustione, caratterizzato da cilindrata (in  $\text{cm}^3$ ) e potenza (in cavalli). Normalmente, due oggetti `Engine` sono uguali se hanno la stessa cilindrata e la stessa potenza. Il metodo `byVolume` converte questo `Engine` in modo che venga confrontata solo la cilindrata. Analogamente, il metodo `byPower` converte questo `Engine` in modo che venga confrontata solo la potenza.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre>Engine a = new Engine(1200, 69), b = new Engine(1200, 75), c =     new Engine(1400, 75); System.out.println(a); System.out.println(a.equals(b));  Engine aVol = a.byVolume(), bVol = b.byVolume(); System.out.println(aVol); System.out.println(aVol.equals(bVol)); System.out.println(a==aVol);  Engine bPow = b.byPower(), cPow = c.byPower(); System.out.println(bPow); System.out.println(bPow.equals(cPow));</pre>	<pre>(1200.0 cm3, 69.0 CV) false (1200.0 cm3, 69.0 CV) true false (1200.0 cm3, 75.0 CV) true</pre>

## 6. (Curriculum, 2016-1-27)

Un oggetto `Curriculum` rappresenta una sequenza di lavori, ognuno dei quali è un'istanza della classe `Job`. Il costruttore di `Curriculum` accetta il nome di una persona. Il metodo `addJob` aggiunge un lavoro alla sequenza, caratterizzato da una descrizione e dall'anno di inizio, restituendo un nuovo oggetto di tipo `Job`. Infine, la classe `Job` offre il metodo `next`, che restituisce *in tempo costante* il lavoro successivo nella sequenza (oppure `null`).

Implementare le classi `Curriculum` e `Job`, rispettando il seguente caso d'uso.

---

Caso d'uso:

```
Curriculum cv = new Curriculum("Walter_White");
Curriculum.Job j1 = cv.addJob("Chimico", 1995);
Curriculum.Job j2 = cv.addJob("Insegnante", 2005);
Curriculum.Job j3 = cv.addJob("Cuoco", 2009);
```

```
System.out.println(j2.next());
System.out.println(j3.next());
```

---

Output:

```
Cuoco: 2009
null
```

---

## 7. (Controller, 2015-6-24)

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
```

---

## 8. (Pizza, 2014-11-3)

[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

## 9. (2010-1-22)

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;
}
```

## 10. (2009-9-1'8)

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) { }
}
```

## 11. (Washer, 2009-7-9)

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>

#### 12. (2009-4-23)

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();
    }
}

```

#### 13. (2009-11-27)

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) { }
}

```

## 14. (Interval, 2009-1-29)

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>

## 15. (Triangolo, 2008-4-21)

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>



# 11 Classi enumerate

## 1. (**LengthUnit**, 2017-1-25)

Realizzare l'enumerazione **LengthUnit**, che rappresenta le principali unità di misura di lunghezza, dei sistemi metrico e imperiale: centimetri (CM), metri (M), chilometri (KM), pollici (INCH), iarde (YARD), e miglia (MILE). Il metodo **convertTo** accetta un'altra unità di misura *u* e un numero in virgola mobile *x*, e converte *x* da questa unità di misura a *u*.

I fattori di conversione per le misure imperiali sono i seguenti: 1 pollice = 0.025 metri, 1 iarda = 0.914 metri, 1 miglio = 1609 metri.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre>System.out.println(LengthUnit.CM.convertTo(LengthUnit.INCH, 10)); System.out.println(LengthUnit.KM.convertTo(LengthUnit.YARD, 3.5) ); System.out.println(LengthUnit.MILE.convertTo(LengthUnit.M, 6.2));</pre>	<pre>3.9370078740157486 3829.3216630196935 9975.800000000001</pre>

## 2. (**NutrInfo**, 2014-7-3)

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:
<pre>NutrInfo x = new NutrInfo(500); x.setNutrient(Nutrient.FAT, 12.0); x.setNutrient(Nutrient.CARBO, 20.0); x.setNutrient(Nutrient.PROTEIN, 15.0); Comparator&lt;NutrInfo&gt; c = NutrInfo.comparatorBy(Nutrient.FAT);</pre>

## 3. (**Status**, 2014-3-5)

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>

## 4. (Pizza, 2014-11-3)

[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

## 5. (Coin, 2014-11-28)

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&lt;Coin&gt; 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>

## 6. (BloodType, 2013-7-9)

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).

## 7. (Note, 2013-12-16)

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:
<pre>Note a = Note.DO;  System.out.println(a.interval(Note.MI)); System.out.println(Note.MI.interval(Note.LA)); System.out.println(Note.LA.interval(Note.SOL));</pre>	<pre>4 5 -2</pre>

#### 8. (NumberType, 2012-7-9)

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

#### 9. (Panino, 2012-4-23)

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>

#### 10. (TetrisPiece, 2010-7-26)

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>

#### 11. (Cardinal, 2009-6-19)

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.

<p>Esempio d'uso:</p> <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>	<p>Output dell'esempio d'uso:</p> <pre> true NNE </pre>
--	---

## 12 Vero o falso

### 1. (2020-2-27)

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

- ☐ ☐ Secondo il pattern **Observer**, gli osservatori devono mantenere un riferimento al soggetto osservato.
- ☐ ☐ Il pattern **Composite** consente di aggregare oggetti in una struttura gerarchica.
- ☐ ☐ Il pattern **Strategy** consente ai client di fornire versioni nuove di un dato algoritmo.
- ☐ ☐ Il pattern **Decorator** consente anche di aggiungere una decorazione ad un oggetto già decorato.
- ☐ ☐ In Java, una classe enumerata può avere costruttori multipli.
- ☐ ☐ In Java, `Set<Object>` è sottotipo di `Collection<?>`.

### 2. (2020-1-24)

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** vieta che lo stesso oggetto sia un osservatore di diversi soggetti.
- ☐ ☐ Nel pattern **Composite** un oggetto primitivo può contenere un altro oggetto primitivo.
- ☐ ☐ Il pattern **Decorator** serve ad aggiungere funzionalità a una classe senza modificarla.
- ☐ ☐ I metodi di accesso posizionale offerti da `List` rappresentano un'istanza del pattern **Iterator**.
- ☐ ☐ In Java, le classi enumerate possono avere campi (attributi) e metodi *custom*.
- ☐ ☐ In Java, `List<String>` è sottotipo di `List<Object>`.

### 3. (2019-9-20)

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 classi enumerate estendono implicitamente la classe `Enum`.
- ☐ ☐ La pre-condizione di un metodo è un obbligo a carico del chiamante.
- ☐ ☐ Secondo il pattern MVC per interfacce grafiche, i Controller interagiscono con i Model.
- ☐ ☐ Nel pattern **Decorator**, oggetto non decorato e oggetto decorato implementano la stessa interfaccia.
- ☐ ☐ Il pattern **Decorator** aggiunge funzionalità ad una classe senza modificarla.
- ☐ ☐ Nel pattern **Strategy** gli oggetti **Strategy** rappresentano varianti di un algoritmo.

### 4. (2019-7-23)

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

- ☐ ☐ `Comparator` è un'interfaccia funzionale.

- ☐ ☐ Un'interfaccia può contenere campi statici.
- ☐ ☐ Il pattern architetturale MVC per interfacce grafiche sfrutta il design pattern Observer.
- ☐ ☐ Il pattern Decorator prevede che il numero di decorazioni possibili sia illimitato.
- ☐ ☐ Il pattern Decorator prevede un metodo per rimuovere una decorazione da un oggetto decorato.
- ☐ ☐ Nel pattern Strategy la classe Context crea un oggetto che è sottotipo di Strategy.

5. (2019-6-24)

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 modificatore volatile dà garanzie di atomicità, visibilità, e ordinamento.
- ☐ ☐ Per poter invocare `x.join()` è necessario possedere il monitor dell'oggetto `x`.
- ☐ ☐ Se `f` e `g` sono due metodi `synchronized`, le chiamate `x.f()` e `x.g()` sono mutuamente esclusive.
- ☐ ☐ Il pattern Decorator consente di applicare una decorazione a un oggetto già decorato.
- ☐ ☐ Il pattern Decorator prevede un metodo per rimuovere una decorazione da un oggetto decorato.
- ☐ ☐ Il pattern Strategy può essere implementato tramite un'enumerazione che elenca le possibili varianti dell'algoritmo.

6. (2019-3-19)

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, più oggetti possono osservare lo stesso oggetto.
- ☐ ☐ Nel pattern Observer, gli osservatori devono mantenere un riferimento all'oggetto osservato.
- ☐ ☐ Il pattern Composite consente a un oggetto composito di contenere altri oggetti compositi.
- ☐ ☐ Il pattern Decorator è un modo di aggiungere funzionalità a un oggetto a runtime.
- ☐ ☐ Una classe enumerata può avere campi privati.

7. (2019-2-15)

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 Strategy prevede che tra le varianti di un algoritmo ce ne sia una di default.
- ☐ ☐ Il pattern Decorator vieta di aggiungere una decorazione ad un oggetto già decorato.
- ☐ ☐ Il pattern Factory Method consente a diverse sottoclassi di creare prodotti concreti diversi.
- ☐ ☐ Nel pattern Factory Method il prodotto generico è sottoclasse del produttore generico.
- ☐ ☐ Una classe enumerata può implementare un'interfaccia.

8. (2019-10-9)

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 enumerata può implementare un'interfaccia.
- ☐ ☐ Una lambda espressione può non avere argomenti.
- ☐ ☐ Un'interfaccia funzionale può avere diversi metodi default, ma deve avere un unico metodo astratto.
- ☐ ☐ La scelta del layout di un componente Swing/AWT è un esempio del pattern Decorator.
- ☐ ☐ Il pattern Decorator permette di aggiungere a una classe quelle funzionalità che non hanno bisogno di nuovi metodi.
- ☐ ☐ Nel pattern Strategy gli oggetti Strategy rappresentano varianti di un algoritmo.

9. (2019-1-23)

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 enumerata può avere più di un costruttore
- ☐ ☐ Il pattern Decorator consente anche di aggiungere una decorazione ad un oggetto già decorato
- ☐ ☐ Il contesto del pattern Strategy prevede che esista un numero predefinito di varianti di un algoritmo
- ☐ ☐ Il contesto del pattern Decorator prevede che un oggetto decorato si possa utilizzare come uno non decorato
- ☐ ☐ I pattern Composite e Decorator sono soluzioni alternative allo stesso problema

10. (2018-9-17)

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 modificatore volatile dà garanzie di atomicità e visibilità, ma non di ordinamento.
- ☐ ☐ Aggiungere un tasto (JButton) ad una finestra AWT rappresenta un'applicazione del pattern Decorator.
- ☐ ☐ Il pattern Strategy si implementa tipicamente utilizzando una classe enumerata.
- ☐ ☐ Il pattern Composite prevede che un contenitore si possa comportare come un oggetto primitivo.
- ☐ ☐ Il pattern Observer consente che lo stesso oggetto sia un osservatore di diversi soggetti.

11. (2018-7-19)

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 vieta che lo stesso oggetto sia un osservatore di diversi soggetti.
- ☐ ☐ Nel pattern Composite oggetto primitivo e oggetto composito implementano una stessa interfaccia.
- ☐ ☐ Il pattern Decorator si applica anche quando il numero di decorazioni possibili è limitato.
- ☐ ☐ I metodi di accesso posizionale offerti da List rappresentano un'istanza del pattern Iterator.
- ☐ ☐ Il pattern Strategy permette ai client di fornire una variante di un algoritmo.

12. (2018-6-20)

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** vieta che lo stesso oggetto sia un osservatore di diversi soggetti.
- ☐ ☐ Il pattern **Observer** prevede che osservatore e soggetto osservato implementino una stessa interfaccia.
- ☐ ☐ Il pattern **Decorator** consente di aggiungere funzionalità a una classe senza modificarla.
- ☐ ☐ I metodi di accesso posizionale offerti da **List** rappresentano un'istanza del pattern **Iterator**.
- ☐ ☐ I design pattern offrono una casistica dei più comuni errori di progettazione.

13. (2018-3-23)

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** e il pattern **Decorator** sono soluzioni alternative allo stesso problema.
- ☐ ☐ Uno dei pre-requisiti del pattern **Strategy** è che non esista un numero predefinito di varianti dell'algoritmo.
- ☐ ☐ Il pattern **Composite** prevede che gli oggetti contenuti conoscano il contenitore in cui sono stati inseriti.
- ☐ ☐ Nel pattern **Decorator**, l'oggetto decorato e quello decoratore implementano una stessa interfaccia.
- ☐ ☐ Il modificatore **volatile** si può applicare a una variabile locale.
- ☐ ☐ Un thread può rimanere in attesa di entrare in un blocco **synchronized** a tempo indefinito.
- ☐ ☐ Ad ogni thread di esecuzione è associato un oggetto della classe **Thread**.
- ☐ ☐ La classe **Thread** offre un metodo per aspettare la terminazione di un altro thread.

14. (2018-2-22)

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 **Factory Method** il produttore generico ha una relazione di dipendenza dal prodotto generico.
- ☐ ☐ L'aggiunta di un pulsante (**JButton**) ad un pannello **AWT** rappresenta un'applicazione del pattern **Strategy**.
- ☐ ☐ Il pattern **Decorator** consente di aggiungere funzionalità ad una classe senza modificarla.
- ☐ ☐ Il pattern **Strategy** serve a migliorare l'efficienza di un algoritmo.
- ☐ ☐ Il pattern **Observer** suggerisce di organizzare gli osservatori in una gerarchia di classi e sotto-classi.

15. (2018-10-18)

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 modificatore **volatile** dà garanzie di atomicità, visibilità, e ordinamento.
- ☐ ☐ Per poter invocare `x.notify()` è necessario possedere il monitor dell'oggetto `x`.
- ☐ ☐ Tipicamente, nel pattern **Decorator**, per decorare un oggetto lo si passa al costruttore di un decoratore.



- ☐ ☐ Nel pattern **Decorator**, i decoratori sono sottoclassi dell'oggetto base non decorato (**ConcreteComponent**).
- ☐ ☐ Il pattern **Composite** richiede che ci sia un metodo per rimuovere un oggetto da un contenitore.
- ☐ ☐ Il pattern **Observer** prevede che il soggetto generatore di eventi conservi un riferimento ai suoi osservatori.

16. (2018-1-24)

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 **Factory Method** il prodotto concreto è sottotipo del prodotto generico.
- ☐ ☐ Aggiungere un tasto (**JButton**) ad una finestra **AWT** rappresenta un'applicazione del pattern **Factory Method**.
- ☐ ☐ Il pattern **Composite** prevede che un contenitore si possa comportare come un oggetto primitivo.
- ☐ ☐ Il pattern **Decorator** consente di aggiungere una ulteriore decorazione ad un oggetto già decorato.

17. (2017-7-20)

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** anche il soggetto osservato implementa l'interfaccia **Observer**
- ☐ ☐ La scelta del layout di un componente **Swing/AWT** rappresenta un'istanza del pattern **Strategy**
- ☐ ☐ Una delle precondizioni del pattern **Factory Method** è che vi siano più tipi di prodotti concreti.
- ☐ ☐ Il metodo **add** dell'interfaccia **Collection** rappresenta un'istanza del pattern **Strategy**.
- ☐ ☐ Nel pattern **Decorator**, tipicamente un oggetto viene decorato passandolo al costruttore di una classe decoratrice.

18. (2017-6-21)

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** gli osservatori interrogano periodicamente il soggetto che genera eventi.
- ☐ ☐ Il pattern **Strategy** suggerisce di usare oggetti per rappresentare varianti di un algoritmo.
- ☐ ☐ Tutte le classi astratte sono applicazioni del pattern **Template Method**.
- ☐ ☐ Il metodo **add** dell'interfaccia **Collection** rappresenta un'istanza del pattern **Factory Method**.
- ☐ ☐ Il pattern **Composite** prevede che si possa iterare sui componenti di un oggetto composto.
- ☐ ☐ Una variabile locale può essere volatile.

19. (2017-3-23)

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 e quello decoratore implementano una stessa interfaccia.
- ☐ ☐ Il pattern **Composite** prevede che il tipo effettivo degli oggetti contenuti (Leaf) sia sottotipo del tipo effettivo dell'oggetto contenitore (Composite).
- ☐ ☐ I design pattern offrono una casistica dei più comuni errori di progettazione.
- ☐ ☐ Il pattern **Composite** e il pattern **Decorator** sono soluzioni alternative allo stesso problema.
- ☐ ☐ Una delle premesse del pattern **Strategy** è che un algoritmo abbia un numero prefissato di varianti.
- ☐ ☐ Una classe interna statica non può essere istanziata se non viene prima istanziata la classe contenitrice.

## 20. (2017-2-23)

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 ha dei riferimenti agli oggetti decoratori.
- ☐ ☐ Il metodo `toString` della classe **Object** rappresenta un esempio del pattern **Factory Method**.
- ☐ ☐ Il pattern **Composite** prevede che gli oggetti primitivi abbiano un riferimento al contenitore in cui sono inseriti.
- ☐ ☐ 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.
- ☐ ☐ Una classe astratta può avere un costruttore.

## 21. (2017-10-6)

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 **Decorator** consente anche di aggiungere una decorazione ad un oggetto già decorato.
- ☐ ☐ Nel pattern **Factory Method** il prodotto generico è sottotipo del produttore generico.
- ☐ ☐ Uno dei pre-requisiti del pattern **Strategy** è che esista un numero predefinito di varianti di un algoritmo.
- ☐ ☐ Il pattern **Composite** prevede che gli oggetti contenuti conoscano il contenitore in cui sono stati inseriti.
- ☐ ☐ Il modificatore `volatile` si può applicare anche a un intero metodo
- ☐ ☐ La classe `LinkedList<T>` è sottotipo di `ArrayList<T>`
- ☐ ☐ La classe `LinkedList<T>` è sottotipo di `List<? extends T>`
- ☐ ☐ La classe `LinkedList<T>` è sottotipo di `List<? super T>`
- ☐ ☐ L'espressione `null instanceof String` ha valore `true`

## 22. (2017-1-25)

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**, il soggetto osservato avvisa gli osservatori degli eventi rilevanti.

- ☐ ☐ Una delle premesse del pattern **Strategy** è che esista un numero predefinito di varianti di un algoritmo.
- ☐ ☐ Nel pattern **Composite**, un oggetto composito può essere vuoto.
- ☐ ☐ Nel pattern **Factory Method**, il prodotto generico è sottotipo del produttore generico.
- ☐ ☐ Il pattern **Decorator** consente di aggiungere funzionalità a una classe senza modificarla.
- ☐ ☐ Una variabile volatile deve essere di tipo primitivo.
- ☐ ☐ L'interfaccia `Map<K,V>` estende `Collection<K>`.
- ☐ ☐ Per ogni oggetto `x`, dovrebbe valere `x.equals(x)==true`.
- ☐ ☐ Le implementazioni di `hashCode` e `equals` nella classe `Object` sono coerenti.
- ☐ ☐ Una volta clonato, un oggetto non dovrebbe più essere modificato.

23. (2016-9-20)

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** gli osservatori interrogano periodicamente il soggetto osservato.
- ☐ ☐ Uno dei pre-requisiti del pattern **Strategy** è che esista un numero predefinito di varianti di un algoritmo.
- ☐ ☐ Il pattern **Factory Method** si applica quando un oggetto deve contenerne altri.
- ☐ ☐ Il pattern **Composite** impedisce di distinguere un oggetto primitivo da uno composito.
- ☐ ☐ Nel pattern **Decorator**, per “decorare” si intende “aggiungere funzionalità”.
- ☐ ☐ Una variabile volatile deve essere di tipo primitivo.

24. (2016-7-21)

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 da decorare ha un metodo che aggiunge una decorazione.
- ☐ ☐ Aggiungere un campo di testo (`JTextView`) ad una finestra `AWT` rappresenta un'applicazione del pattern **Decorator**.
- ☐ ☐ Nel pattern **Strategy**, la classe `Context` ha un metodo che restituisce un oggetto di tipo **Strategy**.
- ☐ ☐ Una delle premesse del pattern **Factory Method** è che i produttori creino prodotti di tipo diverso.
- ☐ ☐ Il modificatore `volatile` si può applicare a campi e metodi.
- ☐ ☐ Il Java Memory Model offre garanzie di performance.

25. (2016-6-22)

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**, il soggetto generatore di eventi ha la responsabilità di notificare gli osservatori.
- ☐ ☐ Aggiungere un tasto (`JButton`) ad una finestra `AWT` rappresenta un'applicazione del pattern **Decorator**.
- ☐ ☐ Nel pattern **Composite**, contenitori e oggetti primitivi implementano la stessa interfaccia.

- ☐ ☐ Nel pattern **Strategy** un oggetto rappresenta una versione di un algoritmo.
- ☐ ☐ Il pattern **Iterator** si applica ogni qual volta si debba svolgere la stessa operazione ripetutamente.

26. (2016-3-3)

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 **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**.
- ☐ ☐ Nel pattern **Decorator** gli oggetti primitivi posseggono un metodo per aggiungere una decorazione.
- ☐ ☐ Il metodo **add** dell'interfaccia **Collection** rappresenta un'istanza del pattern **Template Method**.
- ☐ ☐ **Composite** e **Decorator** hanno diagrammi UML simili, tranne che per la molteplicità di una aggregazione.

27. (2016-1-27)

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 **Factory Method** prevede un prodotto generico e vari prodotti specifici.
- ☐ ☐ Il pattern **Decorator** prevede che oggetti primitivi e decoratori implementino una stessa interfaccia.
- ☐ ☐ Il metodo **Collections.sort** rappresenta un'istanza del pattern **Template Method**.
- ☐ ☐ Secondo il pattern **Observer**, l'oggetto osservato conserva riferimenti ai suoi osservatori.
- ☐ ☐ Il pattern **Strategy** permette di fornire versioni diverse di un algoritmo.

28. (2015-9-21)

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 conserva dei riferimenti agli oggetti decoratori.
- ☐ ☐ Il pattern **Strategy** e il pattern **Observer** sono soluzioni alternative allo stesso problema.
- ☐ ☐ Nel pattern **Composite**, un oggetto composito ne può contenere un altro.
- ☐ ☐ Il pattern **Iterator** consente di esaminare il contenuto di una collezione senza esporre la sua struttura interna.
- ☐ ☐ I design pattern sono soluzioni consigliate per problemi ricorrenti di programmazione.

29. (2015-7-8)

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.

30. (2015-6-24)

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.

31. (2015-2-5)

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**.

32. (2015-1-20)

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.

## 33. (2014-9-18)

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.

## 34. (2014-7-3)

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 sottoclassi.
- ☐ ☐ L'interfaccia `Collection` è un'istanza del pattern `Composite`.
- ☐ ☐ L'interfaccia `Collection` sfrutta il pattern `Iterator`.

## 35. (2014-7-28)

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.

## 36. (2014-3-5)

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`.

37. (2014-11-28)

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.

38. (2014-1-31)

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.

39. (2013-9-25)

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**.

40. (2013-7-9)

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**.

41. (2013-6-25)

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.

42. (2013-3-22)

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.

43. (2013-2-11)

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.

44. (2013-12-16)

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**.

45. (2013-1-22)

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.

46. (2012-9-3)

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.

47. (2012-7-9)

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.

## 48. (2012-6-18)

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.

## 49. (2011-3-4)

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**.

## 50. (2011-2-7)

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.

51. (2010-9-14)

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.

52. (2010-7-26)

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.

53. (2010-6-28)

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.

## 54. (2010-2-24)

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.

## 55. (2010-11-30)

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.

56. (2010-1-22)

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.

57. (2009-9-1'8)

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**.

58. (2009-7-9)

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**.

## 59. (2009-6-19)

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**.

## 60. (2009-2-19)

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.

## 61. (2009-11-27)

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`.

62. (2009-1-29)

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`.

63. (2009-1-15)

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.

## 64. (2008-9-8)

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.

## 65. (2008-7-9)

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.

## 66. (2008-6-19)

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**.

67. (2008-3-27)

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.

68. (2008-2-25)

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>` 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.

69. (2008-1-30)

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.

## 70. (2007-9-17)

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.

## 71. (2007-7-20)

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**.

72. (2007-6-29)

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.

73. (2007-2-7)

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**.

74. (2007-2-23)

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`.

## 75. (2007-1-12)

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.

## 76. (2006-9-15)

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

- ☐ ☐ L'interfaccia `Iterable` ha un parametro di tipo.
- ☐ ☐ La classe `String` ha un parametro di tipo.
- ☐ ☐ Il tipo jolly `"?"` è un parametro attuale di tipo.
- ☐ ☐ Una classe può avere più di un parametro di tipo.
- ☐ ☐ L'interfaccia `List` estende `Collection`.
- ☐ ☐ Le eccezioni derivate da `RuntimeException` non sono verificate.
- ☐ ☐ Gli oggetti di tipo `String` sono immutabili.
- ☐ ☐ Dato un insieme di firme di metodi, non sempre ce ne è una più specifica di tutte le altre.
- ☐ ☐ L'*early binding* è svolto dal programmatore.
- ☐ ☐ Il *late binding* è svolto dalla Java Virtual Machine.

## 77. (2006-7-17)

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.

78. (2006-6-26)

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.



## 13 Clonazione di oggetti

### 1. (Book, 2016-7-21)

Implementare la classe `Book`, che rappresenta un libro diviso in capitoli. Il metodo `addChapter` aggiunge un capitolo in coda al libro, caratterizzato da titolo e contenuto. I capitoli sono automaticamente numerati a partire da 1. Il metodo `getChapterName(i)` restituisce il titolo del capitolo *i*-esimo, mentre il metodo `getChapterContent(i)` ne restituisce il contenuto.

Gli oggetti `Book` devono essere clonabili. Inoltre, la classe deve essere dotata di ordinamento naturale, basato sul numero di capitoli.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre>Book b = new Book(); b.addChapter("Prefazione", "Sono passati pochi anni..."); b.addChapter("Introduzione", "Un_calcolatore_digitale..."); ; b.addChapter("Sistemi_di_elaborazione", "Un_calcolatore..."); Book bb = b.clone(); System.out.println(bb.getChapterContent(1)); System.out.println(bb.getChapterTitle(2));</pre>	<pre>Sono passati pochi anni... Introduzione</pre>

### 2. (Insieme di lettere, 2013-1-22)

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>

### 3. (Anagrammi, 2012-9-3)

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>

## 4. (Segment, 2010-11-30)

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>

## 5. (2010-1-22)

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;
}

```

## 6. (TreeType, 2006-9-15)

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>



## 14 Riflessione

### 1. (2022-1-26)

In riferimento all'esercizio 1, dire quali dei seguenti sono criteri di uguaglianza validi tra oggetti di tipo `Exchange`. In caso negativo, dire quale o quali proprietà sono violate e descrivere un controesempio. Due oggetti `Exchange` sono uguali se:

- a) Contengono gli stessi titoli, anche con valori diversi.
- b) Tutti i titoli in comune hanno lo stesso valore (ma potrebbero avere anche titoli diversi).
- c) Almeno uno dei due `Exchange` ha un titolo con almeno 2 alert.

### 2. (2022-1-26)

Determinare l'output del seguente frammento di codice Java:

```
Short s = 10;
Integer a = 10;
Float f = new Float(10f);
Number aa = a;
Object o = f;

System.out.println((Object)s == (Object)a);
System.out.println((s+1) == (a+1));
System.out.println(aa == a);
System.out.println(a.getClass().equals(aa.getClass()));
System.out.println(o.getClass().equals(f.getClass()));
System.out.println(o.getClass().equals(Object.class));
```

### 3. (2021-10-26)

Determinare l'output del seguente frammento di codice Java:

```
Short s = 10;
Integer a = 10;
Double b = 11.0;
Number aa = a;
Number bb = b;

System.out.println( ((Object) s) == ((Object) a) );
System.out.println(aa == a);
System.out.println(a.getClass().equals(aa.getClass()));
System.out.println(aa.getClass().equals(bb.getClass()));
Object o = s;
System.out.println(o.getClass().equals(Short.class));
```

### 4. (GetByType, 2010-1-22)

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.

### 5. (CountByType, 2009-11-27)

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&lt;Number&gt; l = <b>new</b> LinkedList&lt;Number&gt;(); l.add(<b>new</b> Integer(3)); l.add(<b>new</b> Double(4.0)) l.add(<b>new</b> Float(7.0f)); l.add(<b>new</b> Integer(11)); countByType(l);</pre>	<pre>java.lang.Double : 1 java.lang.Float : 1 java.lang.Integer : 2</pre>

#### 6. (SuperclassIterator, 2006-9-15)

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.

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

# 15 Multi-threading

## 1. (MysteryThread6, 2021-9-24)

Escludendo i cosiddetti *spurious wakeup*, elencare tutte le sequenze di output possibili per il seguente programma.

```
public static void main(String[] args) throws InterruptedException {
    final Object x = new Object();
    final int[] count = new int[1];

    class MyThread extends Thread {
        int id;
        MyThread(int n) { id = n; }
        public void run() {
            synchronized (x) {
                count[0]++;
                synchronized (count) {
                    count.notify();
                }
                try {
                    x.wait();
                } catch (Exception e) { }
            }
            System.out.println(id);
        }
    }
    Thread t1 = new MyThread(1), t2 = new MyThread(2), t3 = new MyThread(3);
    t1.start(); t2.start(); t3.start();
    synchronized (count) {
        while (count[0]<3) {
            count.wait();
            System.out.println("Incremento");
        }
    }
    System.out.println("Fatto");
    synchronized (x) {
        x.notifyAll();
    }
}
```

## 2. (2021-7-26)

Il seguente thread accede ad una lista di Employee, precedentemente istanziata. Gli oggetti Employee hanno un campo salario (*salary*) e un campo anzianità in servizio (*years*). Un thread di tipo AgeBonus assegna un bonus di 100 euro agli impiegati la cui anzianità supera una data soglia.

```
public class AgeBonus extends Thread {
    private final int threshold;
    public AgeBonus(int n) { this.threshold = n; }

    @Override
    public void run() {
        -----1-----
        for (Employee e: list) {
            -----2-----
        }
    }
}
```

```

        if (e.getYears() > threshold)
            e.setSalary(e.getSalary() + 100);
        -----3-----
    }
    -----4-----
}

```

Dire quali dei seguenti inserimenti consentono a un programma di eseguire concorrentemente un numero arbitrario di thread **AgeBonus**, senza incorrere in *race condition* (è possibile indicare più risposte, intese come alternative).

In aggiunta, dire quale inserimento è quello migliore e perché.

- |     |                                     |   |
|-----|-------------------------------------|---|
| (a) | 1 = “ <b>synchronized (this){</b> ” | 4 = “ <b>}</b> ”  |
| (b) | 1 = “ <b>synchronized {</b> ”       | 4 = “ <b>}</b> ”  |
| (c) | 1 = “ <b>synchronized (list){</b> ” | 4 = “ <b>}</b> ”  |
| (d) | 2 = “ <b>synchronized (this){</b> ” | 3 = “ <b>}</b> ”  |
| (e) | 2 = “ <b>synchronized (list){</b> ” | 3 = “ <b>}</b> ”  |
| (f) | 2 = “ <b>synchronized (e){</b> ”    | 3 = “ <b>}</b> ”  |
| (g) | 2 = “list.wait();”                  | 3 = “list.notifyAll();”   |
| (h) | 1 = “ <b>synchronized(list){</b> ”  | 2 = “list.wait();”      3 = “list.notifyAll();”      4 = “ <b>}</b> ” |

### 3. (Missing synch 3, 2021-10-26)

Le istanze della seguente classe **MyThread** condividono due array di interi (**int[]**), **a** e **b**, precedentemente istanziati e inizializzati.

```

class MyThread extends Thread {
    public void run() {
        -----1-----
        for (int i=0; i<a.length; i++) {
            -----2-----
            if (a[i] > b[i]) {
                int temp = b[i];
                b[i] = a[i];
                a[i] = temp;
            }
            -----3-----
        }
        -----4-----
    }
}

```

Un programma avvia *due* thread di tipo **MyThread**, con l’obiettivo che, dopo l’esecuzione, ciascun elemento di **a** sia minore o uguale del corrispondente elemento di **b**.

Dire quali dei seguenti inserimenti rendono il programma corretto ed esente da *race condition* (è possibile indicare più risposte):

- |     |   |
|-----|---|
| (a) | non è necessario aggiungere nulla                                   |
| (b) | 1 = “ <b>synchronized (this){</b> ”      4 = “ <b>}</b> ”           |
| (c) | 1 = “ <b>synchronized (MyThread.class){</b> ”      4 = “ <b>}</b> ” |
| (d) | 1 = “ <b>synchronized {</b> ”      4 = “ <b>}</b> ”                 |
| (e) | 1 = “ <b>synchronized (a){</b> ”      4 = “ <b>}</b> ”              |
| (f) | 1 = “ <b>synchronized (b){</b> ”      4 = “ <b>}</b> ”              |
| (g) | 2 = “ <b>synchronized (this){</b> ”      3 = “ <b>}</b> ”           |
| (h) | 2 = “ <b>synchronized (a[i]){</b> ”      3 = “ <b>}</b> ”           |
| (i) | 2 = “ <b>synchronized (b){</b> ”      3 = “ <b>}</b> ”              |
| (j) | 2 = “a.wait();”      3 = “a.notifyAll();”                           |

### 4. (MysteryThread8, 2020-2-27)

Elencare tutte le sequenze di output possibili per il seguente programma.

```

public class A {
    private volatile int n;
    public int incrementAndGet() {
        return ++n;
    }

    public static void main(String[] args) {
        A a = new A(), b = new A();
        Thread t1 = new Thread(() -> System.out.println(a.incrementAndGet())),
            t2 = new Thread(() -> System.out.println(b.incrementAndGet())),
            t3 = new Thread(() -> System.out.println(a.incrementAndGet()));
        t1.start(); t2.start(); t3.start();
    }
}

```

5. (MysteryThread7, 2020-1-24)

Escludendo i cosiddetti *spurious wakeup*, elencare tutte le sequenze di output possibili per il seguente programma.

```

public static void main(String[] args) throws InterruptedException {
    final Object x = new Object();
    final int[] count = new int[1]; // Don't do this: use AtomicInteger

    class MyThread extends Thread {
        public void run() {
            synchronized (x) {
                count[0]++;
                x.notify();
                System.out.println(count[0]);
            }
        }
    }

    Thread t1 = new MyThread(), t2 = new MyThread(), t3 = new MyThread();
    t1.start(); t2.start(); t3.start();
    synchronized (x) {
        count[0] = -1;
        while (count[0] < 0) x.wait();
    }
    t2.join();
    System.out.println("Fatto");
}

```

6. (MysteryThread6, 2019-9-20)

(a) Escludendo i cosiddetti *spurious wakeup*, elencare tutte le sequenze di output possibili per il seguente programma.

```

public static void main(String[] args) throws InterruptedException {
    final Object x = new Object(), y = new Object();
    Thread t1 = new Thread(() -> {
        synchronized (x) {
            try {
                x.wait();
            }
            synchronized (y) {
                y.notify();
            }
        }
        catch (Exception e) { return; }
        finally { System.out.println("t1"); }
    });
    Thread t2 = new Thread(() -> {
        synchronized (y) {

```

```

        try { y.wait(); }
        catch (Exception e) { return; }
        finally { System.out.println("t2"); }
    }
});
t1.start();
t2.start();
synchronized (y) {
    y.notify();
}
System.out.println("main");
}

```

(b) Come cambiano le sequenze di output se le ultime righe del `main` vengono modificate come segue?

```

synchronized (y) {
    y.notify();
    System.out.println("main");
}
}

```

#### 7. (RandomExecutor, 2019-7-23)

Implementare la classe `RandomExecutor` che esegue degli oggetti `Runnable` sequenzialmente e in ordine casuale. Precisamente, la classe offre un costruttore senza argomenti e i seguenti metodi:

- Il metodo `void addTask(Runnable r)` aggiunge un task. Questo metodo può essere chiamato solo prima di `launch`.
- Il metodo `void launch()` avvia l'esecuzione dei task. Questo metodo non è bloccante.
- Il metodo bloccante `void join(Runnable r)` attende la terminazione del corrispondente task. Questo metodo può essere chiamato prima o dopo `launch`.

L'implementazione deve rispettare il seguente esempio d'uso.

<p>Esempio d'uso:</p> <pre> RandomExecutor e = new RandomExecutor(); Runnable r1 = () -&gt; { System.out.println(1); }; Runnable r2 = () -&gt; { System.out.println(2); }; Runnable r3 = () -&gt; { System.out.println(3); }; e.addTask(r1); e.addTask(r2); e.addTask(r3); e.launch(); e.join(r2); </pre>	<p>Output:</p> <p>I numeri da 1 a 3 in ordine casuale.</p>
---	--

#### 8. (Missing synch 3, 2019-3-19)

Le istanze di `MyThread` condividono un array di interi `a` (tipo `int[]`), precedentemente istanziato e inizializzato.

```

class MyThread extends Thread {
    public void run() {
        final int n = a.length - 1;
        1
        for (int i=0; i <= n/2; i++) {
            2
            // scambia a[i] e a[n-i]
            int temp = a[i];
            a[i] = a[n - i];
            a[n - i] = temp;
        }
    }
}

```



```

        } -----3-----
    } -----4-----
}

```

Se un programma avvia *un solo* thread di tipo `MyThread`, questo rovescerà il contenuto dell'array `a`.

Un programma avvia *due* thread di tipo `MyThread`, con l'intento di ritrovare l'array `a` inalterato. Dire quali dei seguenti inserimenti rendono il programma corretto ed esente da *race condition* (è possibile indicare più risposte):

- |     |   |                                |
|-----|---|--------------------------------|
| (a) | non c'è bisogno di aggiungere nulla                                       |                                |
| (b) | aggiungere il modificatore <b>synchronized</b> al metodo <code>run</code> |                                |
| (c) | <code>1 = "synchronized (this){"</code>                                   | <code>4 = "}"</code>           |
| (d) | <code>1 = "synchronized {"</code>   | <code>4 = "}"</code>           |
| (e) | <code>1 = "synchronized (a){"</code>                                      | <code>4 = "}"</code>           |
| (f) | <code>1 = "synchronized (MyThread.class){"</code>                         | <code>4 = "}"</code>           |
| (g) | <code>1 = "a.wait();"</code>  | <code>4 = "a.notify();"</code> |
| (h) | <code>2 = "synchronized (this){"</code>                                   | <code>3 = "}"</code>           |
| (i) | <code>2 = "synchronized (a[i]){"</code>                                   | <code>3 = "}"</code>           |
| (j) | <code>2 = "synchronized (a){"</code>                                      | <code>3 = "}"</code>           |
| (k) | <code>2 = "synchronized (MyThread.class){"</code>                         | <code>3 = "}"</code>           |

#### 9. (Missing synch 2, 2019-2-15)

Le istanze di `MyThread` condividono due array di interi, `a` e `b`, precedentemente istanziati e inizializzati.

```

class MyThread extends Thread {
    public void run() {
        -----1-----
        for (int i=0; i<a.length; i++) {
            -----2-----
            if (a[i]>0) {
                b[i] = a[i];
                a[i] = 0;
            }
            -----3-----
        }
        -----4-----
    }
}

```

Un programma avvia due thread di tipo `MyThread`, con l'obiettivo di spostare i valori positivi da `a` a `b`.

Dire quali dei seguenti inserimenti rendono il programma corretto ed esente da *race condition* (è possibile indicare più risposte):

- |     |   |                                |
|-----|---|--------------------------------|
| (a) | non c'è bisogno di aggiungere nulla     |                                |
| (b) | <code>1 = "synchronized (this){"</code> | <code>4 = "}"</code>           |
| (c) | <code>1 = "synchronized {"</code>       | <code>4 = "}"</code>           |
| (d) | <code>1 = "synchronized (a){"</code>    | <code>4 = "}"</code>           |
| (e) | <code>2 = "synchronized (this){"</code> | <code>3 = "}"</code>           |
| (f) | <code>2 = "synchronized (a[i]){"</code> | <code>3 = "}"</code>           |
| (g) | <code>2 = "synchronized (b){"</code>    | <code>3 = "}"</code>           |
| (h) | <code>2 = "a.wait();"</code>            | <code>3 = "a.notify();"</code> |

#### 10. (Shop, 2019-10-9)

Implementare la classe thread-safe `Shop<T>` che rappresenta un negozio di oggetti di tipo `T`, e fornisce i seguenti metodi:

- **public void** `sell(T object, int price)`  
Mette in vendita un oggetto ad un dato prezzo
- **public T** `buy(int offer)`  
Mette in attesa il chiamante finché non ci sia un oggetto in vendita il cui prezzo è inferiore o uguale all'offerta; a quel punto rimuove quell'oggetto dalla vendita e lo restituisce al chiamante

Nota: se viene invocato due volte `sell` con lo stesso oggetto, senza che sia stato venduto nel frattempo, il secondo prezzo deve sostituire il primo.

11. (**GuessTheNumber, 2019-1-23**)

Realizzare la classe `GuessTheNumber`, che consente a diversi thread di indovinare un numero segreto. Il costruttore accetta il numero da indovinare (`int`) e la durata del quiz, in millisecondi (`long`). Il metodo `guessAndWait` consente ad un thread di proporre una soluzione (`int`), poi attende fino alla fine del quiz, ed infine restituisce `true` se questo è il thread che si è avvicinato di più alla soluzione, e `false` altrimenti.

Rispondere alle seguenti domande relative alla vostra implementazione:

- Cosa succede se più thread arrivano a pari merito?
- Cosa succede se un thread chiama `guessAndWait` quando il quiz è già terminato?

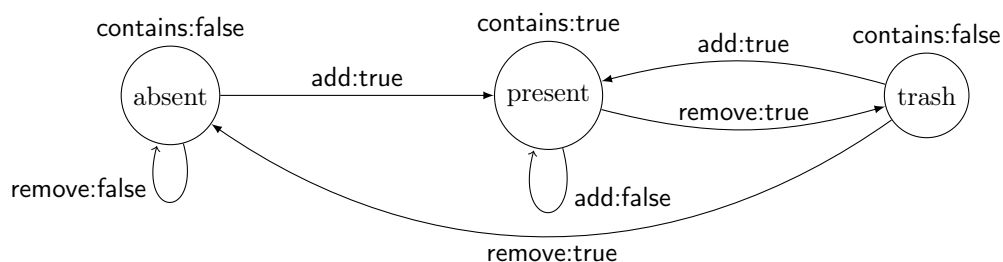
12. (**SharedCounter, 2018-9-17**)

Realizzare la classe `SharedCounter`, che rappresenta un contatore *thread-safe*, che parte dal valore zero. Il suo metodo `incr` incrementa il contatore, mentre `decr` lo decrementa. Il contatore non può assumere un valore negativo, quindi invocare `decr` quando il valore è zero non ha alcun effetto.

Il metodo `waitForValue` accetta un valore intero `n` e mette il chiamante in attesa finché il contatore non assuma il valore `n` (se il contatore vale già `n`, il metodo restituisce subito il controllo al chiamante).

13. (**SafeSet, 2018-7-19**)

Realizzare la classe `SafeSet`, che rappresenta un insieme che richiede due passaggi per rimuovere completamente un oggetto. Il metodo `add` aggiunge un elemento all'insieme, restituendo `true` se l'inserimento ha avuto successo. Il metodo `remove` rimuove un elemento dall'insieme, ma la rimozione è definitiva solo dopo una seconda chiamata. Il metodo `contains` verifica se l'insieme contiene un dato elemento (in base a `equals`). Infine, un `SafeSet` deve essere *thread-safe*. Il seguente diagramma rappresenta il ciclo di vita di un oggetto all'interno di un `SafeSet`:



L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre>SafeSet&lt;String&gt; a = new SafeSet&lt;&gt;(); System.out.println(a.add("ciao")); System.out.println(a.add("mondo")); System.out.println(a.remove("ciao")); System.out.println(a.contains("ciao")); System.out.println(a.remove("ciao")); System.out.println(a.contains("ciao"));</pre>	<pre>true true true false true false</pre>

#### 14. (MysteryThread5, 2018-7-19)

Escludendo i cosiddetti *spurious wakeup*, elencare tutte le sequenze di output possibili per il seguente programma.

```
public static void main(String[] args) throws InterruptedException {
    final Object x = new Object();
    final int[] count = new int[1]; // don't do this: use AtomicInteger

    class MyThread extends Thread {
        int id;
        MyThread(int n) { id = n; }
        public void run() {
            synchronized (x) {
                synchronized (count) {
                    count[0]++;
                    count.notify();
                }
            }
            try {
                x.wait();
            } catch (Exception e) {
                x.notify();
            } finally {
                System.out.println(id);
            }
        }
    }

    Thread t1 = new MyThread(1), t2 = new MyThread(2), t3 = new MyThread(3);
    t1.start(); t2.start(); t3.start();
    synchronized (count) {
        while (count[0] < 3) count.wait();
    }
    t2.interrupt();
    t2.join();
    System.out.println("Fatto");
}
```

#### 15. (PeriodicExecutor, 2018-6-20)

Realizzare la classe `PeriodicExecutor`, che consente di eseguire una serie di task periodicamente, con un limite al numero di task che possono essere eseguiti contemporaneamente. Il costruttore accetta questo limite. Il metodo `addTask` accetta un `Runnable` e un `long x`, e fa in modo che il `Runnable` venga eseguito ripetutamente, con un ritardo di  $x$  millisecondi tra la fine di un'esecuzione e l'inizio della successiva.

Se però (ri)avviare un `Runnable` porta a superare il limite, l'avvio viene rimandato finché ci sarà la possibilità di eseguirlo senza violare il limite.

Il limite si riferisce al numero di task che stanno eseguendo il loro `Runnable`, non al periodo durante il quale stanno aspettando il ritardo  $x$ .

L'implementazione deve rispettare il seguente esempio d'uso.

```
PeriodicExecutor exec = new PeriodicExecutor(2);
Runnable r1 = ..., r2 = ..., r3 = ...;
exec.addTask(r1, 1000);
exec.addTask(r2, 500);
exec.addTask(r3, 700);
```

#### 16. (Two threads, 2018-2-22)

Implementare un programma Java che avvia contemporaneamente due thread, che condividono una lista di interi:

- a) il primo thread aggiunge un numero casuale alla lista ogni decimo di secondo, terminando quando il numero estratto è multiplo di 100;
- b) il secondo thread calcola e stampa a video la somma di tutti i numeri nella lista, una volta al secondo, terminando non appena termina il primo thread.

È necessario evitare *race condition* e attese attive.

*Suggerimento.* Per ottenere un numero intero casuale, si consiglia di utilizzare le seguenti funzionalità della classe `java.util.Random`:

```
public Random()
public int nextInt()
```

#### 17. (Shared total, 2018-1-24)

Implementare la classe `SharedTotal`, che permette a diversi thread di comunicare un valore numerico (`double`) e poi ricevere la somma di tutti i valori inviati dai diversi thread. Precisamente, il costruttore accetta come argomento un *timeout* in millisecondi. Il metodo `sendValAndReceiveTot` accetta come argomento il valore indicato dal thread corrente, mette il thread corrente in attesa fino allo scadere del timeout, e infine restituisce il totale di tutti i valori inviati.

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

È necessario evitare *race condition*.

Esempio d'uso:  <pre>SharedTotal tot = new SharedTotal(1000); try {     System.out.println(tot.sendValAndReceiveTot(5.0)); } catch (InterruptedException e) {     ... }</pre>	Comportamento: In assenza di altri thread, dopo un secondo stamperà 5.0
---	--

#### 18. (MysteryThread4, 2017-7-20)

Escludendo i cosiddetti *spurious wakeup*, 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 Object object;
        public MyThread(int n, Object x) {
            id = n;
            object = x;
        }
        public void run() {
            try {
                synchronized (object) {
                    object.wait();
                }
            } catch (InterruptedException e) {
                return;
            }
            System.out.println(id);
        }
    }
    Object o1 = new Object(), o2 = new Object();
    Thread t1 = new MyThread(1,o1);
    Thread t2 = new MyThread(2,o1);
    Thread t3 = new MyThread(3,o1);
```

```

Thread t4 = new MyThread(4,o2);
t1.start(); t2.start(); t3.start(); t4.start();
try { Thread.sleep(1000); } catch (InterruptedException e) { }
synchronized (o2) { o2.notifyAll(); }
synchronized (o1) { o1.notify(); }
}

```

#### 19. (Market, 2017-6-21)

Realizzare la classe *thread-safe* **Market**, che permette a diversi thread di contrattare il prezzo di un servizio. Il metodo **sell** offre il servizio per un certo prezzo e blocca il chiamante finché non arriverà una richiesta con un importo adeguato. Simmetricamente, il metodo **buy** richiede il servizio per un certo prezzo e blocca il chiamante finché non arriverà un'offerta adeguata.

Per semplicità, si può supporre che tutti gli importi passati a **sell** e **buy** siano diversi.

L'implementazione deve rispettare il seguente esempio d'uso, in cui diversi thread condividono un oggetto **Market** **m**:

Thread 1:	m.buy(10.0);	resta bloccato
Thread 2:	m.sell(15.50);	resta bloccato
Thread 3:	m.sell(12.0);	resta bloccato
Thread 4:	m.buy(13.0);	sblocca T3 e ritorna
Thread 5:	m.buy(11.0);	resta bloccato
Thread 6:	m.sell(9.50);	sblocca T1 oppure T5 e ritorna

#### 20. (Bonus per Employee, 2017-3-23)

I seguenti thread accedono ad una lista di **Employee**, precedentemente istanziata. Gli oggetti **Employee** hanno un campo salario (*salary*) e un campo anzianità in servizio (*years*). Il primo thread assegna un bonus di 150 agli impiegati in servizio da più di 10 anni. Il secondo thread assegna un bonus di 200 agli impiegati che hanno un salario inferiore a 1500.

<pre> class AgeBonus extends Thread {     public void run() {         1         for (Employee e: list) {             2             if (e.getYears()&gt;10)                 e.setSalary(e.getSalary()+150);             3         }         4     } } </pre>	<pre> class LowBonus extends Thread {     public void run() {         1         for (Employee e: list) {             2             if (e.getSalary()&lt;1500)                 e.setSalary(e.getSalary()+200);             3         }         4     } } </pre>
---	--

Se un programma avvia questi due thread, dire quali dei seguenti inserimenti rendono il programma corretto ed esente da *race condition* (è possibile indicare più risposte, intese come alternative).

Inoltre, se si sceglie più di una risposta, commentare sinteticamente la performance che si otterrebbe con ciascuna di esse.

- (a) 1 = "**synchronized (this){**"    4 = "**}**"
- (b) 1 = "**synchronized {**"    4 = "**}**"
- (c) 1 = "**synchronized (list){**"    4 = "**}**"
- (d) 2 = "**synchronized (this){**"    3 = "**}**"
- (e) 2 = "**synchronized (list){**"    3 = "**}**"
- (f) 2 = "**synchronized (e){**"    3 = "**}**"
- (g) 2 = "list.wait();"    3 = "list.notify();"
- (h) 1 = "**synchronized {**"    2 = "list.wait();"    3 = "list.notify();"    4 = "**}**"

21. (**sumAndMax, 2017-2-23**)

Implementare il metodo statico `sumAndMax`, che accetta un array di `double` e restituisce un array di due `double`, che conterranno rispettivamente la somma e il massimo del primo array.

Il metodo deve calcolare i due risultati in parallelo. Inoltre, qualora la somma (anche parziale) andasse in overflow, il metodo deve interrompere il thread che sta calcolando il massimo e restituire `null`.

*Suggerimento:* se un'addizione tra due `double` va in overflow, il suo risultato sarà `Double.POSITIVE_INFINITY` o `Double.NEGATIVE_INFINITY`.

22. (**Somma e azzera, 2017-10-6**)

Il seguente thread accede ad un array di interi, precedentemente istanziato.

```
class MyThread extends Thread {
    public void run() {
        int tot = 0;
        -----1-----
        for (int i=0; i<array.length; i++) {
            -----2-----
            tot += array[i];
            array[i] = 0;
            -----3-----
        }
        -----4-----
        System.out.println(tot);
    }
}
```

Un programma avvia due thread di tipo `MyThread`, con l'obiettivo di ottenere l'output

```
<totale dell'array>
0
```

Dire quali dei seguenti inserimenti garantiscono l'output desiderato (è possibile indicare più risposte):

- (a) 1 = "`synchronized (this)`"    4 = "`}`"
- (b) 1 = "`synchronized` {"    4 = "`}`"
- (c) 1 = "`synchronized (array)`"    4 = "`}`"
- (d) 1 = "`synchronized (tot)`"    4 = "`}`"
- (e) 2 = "`synchronized (this)`"    3 = "`}`"
- (f) 2 = "`synchronized (array)`"    3 = "`}`"

23. (**mergeIfSorted, 2017-1-25**)

Implementare il metodo statico `mergeIfSorted`, che accetta due liste *a* e *b*, e un comparatore *c*, e restituisce un'altra lista. Inizialmente, usando due thread diversi, il metodo verifica che le liste *a* e *b* siano ordinate in senso non decrescente (ogni thread si occupa di una lista). Poi, se le liste sono effettivamente ordinate, il metodo le fonde (senza modificarle) in un'unica lista ordinata, che viene restituita al chiamante. Se, invece, almeno una delle due liste non è ordinata, il metodo termina restituendo `null`.

Il metodo dovrebbe avere complessità di tempo lineare.

Porre particolare attenzione alla scelta della firma, considerando i criteri di funzionalità, completezza, correttezza, fornitura di garanzie e semplicità.

24. (**Somma due, 2016-9-20**)

Il seguente thread accede ad un array di interi, precedentemente istanziato.

```
class MyThread extends Thread {
    public void run() {
        -----1-----
```

```

    for (int i=0; i<array.length; i++) {
        -----2-----
        array[i]++;
        -----3-----
    }
    -----4-----
}

```

Un programma avvia due thread di tipo `MyThread`, con l'obiettivo di incrementare ogni elemento dell'array di 2. Dire quali dei seguenti inserimenti rendono il programma corretto ed esente da *race condition* (è possibile indicare più risposte):

- (a) 1 = "**synchronized (this)** {"      4 = {"}"
- (b) 1 = "**synchronized** {"              4 = {"}"
- (c) 1 = "**synchronized** (array) {"      4 = {"}"
- (d) 2 = "**synchronized (this)** {"      3 = {"}"
- (e) 2 = "**synchronized** (array) {"      3 = {"}"
- (f) 2 = "array.wait();"                  3 = "array.notify();" "

#### 25. (**findString**, 2016-7-21)

Implementare il metodo statico `findString` che accetta una stringa  $x$  e un array di stringhe  $a$  e restituisce "vero" se  $x$  è una delle stringhe di  $a$ , e "falso" altrimenti. Per ottenere questo risultato, il metodo usa due tecniche in parallelo: un primo thread confronta  $x$  con ciascuna stringa dell'array; un altro thread confronta solo la lunghezza di  $x$  con quella di ciascuna stringa dell'array. Il metodo deve restituire il controllo al chiamante appena è in grado di fornire una risposta certa.

Ad esempio, se il secondo thread scopre che nessuna stringa dell'array ha la stessa lunghezza di  $x$ , il metodo deve subito restituire "falso" e terminare il primo thread (se ancora in esecuzione).

#### 26. (**BlockingArray**, 2016-6-22)

Realizzare la classe `BlockingArray`, che rappresenta un array di dimensione fissa, in cui diversi thread mettono e tolgono elementi.

Il costruttore accetta la dimensione dell'array. Inizialmente, tutte le celle risultano vuote. Il metodo `put(i, x)` inserisce l'oggetto  $x$  nella cella  $i$ -esima, aspettando se quella cella è occupata. Simmetricamente, il metodo `take(i)` preleva l'oggetto dalla cella  $i$ -esima, aspettando se quella cella è vuota. La classe deve risultare *thread-safe*.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre> BlockingArray&lt;String&gt; array = new BlockingArray     &lt;&gt;(10); array.put(0, "uno"); array.put(1, "due"); System.out.println(array.take(0)); array.put(1, "tre"); </pre>	<pre> uno  A questo punto il thread si ferma finché un altro thread non invocherà take(1) </pre>

#### 27. (**MysteryThread3**, 2016-3-3)

Escludendo i cosiddetti *spurious wakeup*, 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 Object object;
        public MyThread(int n, Object x) {
            id = n;

```

```

        object = x;
    }
    public void run() {
        try {
            if (object!=null) synchronized (object) {
                object.wait();
            }
        } catch (InterruptedException e) { return; }
        System.out.println(id);
    }
}
Object o1 = new Object(), o2 = new Object();
Thread t1 = new MyThread(1,null);
Thread t2 = new MyThread(2,o1);
Thread t3 = new MyThread(3,o1);
Thread t4 = new MyThread(4,o2);
t1.start(); t2.start(); t3.start(); t4.start();
try { Thread.sleep(1000); } catch (InterruptedException e) { }
synchronized (o2) { o2.notify(); }
synchronized (o1) { o1.notify(); }
}

```

28. (**twoPhases, 2016-1-27**)

Implementare il metodo statico `twoPhases`, che accetta due `Iterable<Runnable>` ed esegue in parallelo tutti i `Runnable` contenuti nel primo `Iterable` (prima fase), seguiti da tutti i `Runnable` contenuti nel secondo `Iterable` (seconda fase). Precisamente, appena l'*i*-esimo `Runnable` del primo gruppo termina, quel thread passa ad eseguire l'*i*-esimo `Runnable` del secondo gruppo.

29. (**StringQuiz, 2015-9-21**)

La classe `StringQuiz` consente a diversi thread di tentare di indovinare una stringa segreta, entro un tempo prestabilito. Il costruttore accetta la stringa segreta e un timeout in millisecondi. Il metodo `guess` accetta una stringa e restituisce vero se è uguale a quella segreta e falso altrimenti. Ciascun thread ha 3 possibilità di indovinare, oltre le quali il metodo `guess` lancia un'eccezione. Scaduto il timeout, il metodo `guess` lancia un'eccezione ogni volta che viene invocato. La classe deve risultare thread-safe.

30. (**TimeToFinish, 2015-7-8**)

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

```

---



31. (**SimpleThread**, 2015-6-24)

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);
    }
}
```

32. (**ForgetfulSet**, 2015-2-5)

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
```

---

33. (**atLeastOne**, 2014-9-18)

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.

34. (**Exchanger**, 2014-7-3)

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

### 35. (PriorityExecutor, 2014-7-28)

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

### 36. (PeriodicTask, 2014-3-5)

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

### 37. (Alarm, 2014-11-28)

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>	<b>Allarme!</b>

### 38. (PostOfficeQueue, 2014-1-31)

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:
<pre>PostOfficeQueue poq = new PostOfficeQueue(5); System.out.println(poq.getFreeDesk()); poq.deskStart(0); System.out.println(poq.getFreeDesk()); poq.deskStart(1); poq.deskStart(2); System.out.println(poq.getFreeDesk());</pre>	<pre>0 1 3</pre>

### 39. (executeWithDeadline, 2013-9-25)

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.

### 40. (processArray, 2013-7-9)

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&lt;String&gt; f = new RunnableFunction&lt;String&gt;() {     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>

#### 41. (MultiBuffer, 2013-6-25)

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&lt;Integer&gt; mb = new MultiBuffer&lt;Integer&gt;(3); mb.insert(13); mb.insert(24); mb.insert(35); System.out.println(mb.take(0));</pre>	<pre>13</pre>

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:

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

#### 42. (Shared average, 2013-3-22)

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*.

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

#### 43. (Concurrent filter, 2013-2-11)

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`).

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

#### 44. (concurrentMax, 2013-12-16)

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.

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

#### 45. (Shared object, 2013-1-22)

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();
}

```

## 46. (Mystery thread 2, 2012-9-3)

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();
}

```

## 47. (Mystery thread, 2012-7-9)

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");
}

```

#### 48. (ThreadRace, 2012-6-18)

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.

#### 49. (MultiProgressBar, 2011-3-4)

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.

#### 50. (VoteBox, 2011-2-7)

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>

51. (**ExecuteInParallel, 2010-9-14**)

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.

52. (**QueueOfTasks, 2010-6-28**)

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>

53. (**Auction, 2009-9-18**)

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>	<pre> Oggetto venduto a Giulia per 1500 euro. </pre>



#### 54. (Elevator, 2009-7-9)

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 = <b>new</b> 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)

#### 55. (RunnableWithArg, 2008-9-8)

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:
Set<Integer> s = <b>new</b> HashSet<Integer>();	
s.add(3); s.add(13); s.add(88);	1
	6
RunnableWithArg<Integer> r = <b>new</b> RunnableWithArg<Integer>() {	44
<b>public void</b> run(Integer i) {	
System.out.println(i/2);	
}	
};	
Thread t = <b>new</b> RunOnSet<Integer>(r, s);	
t.start();	

#### 56. (MutexWithLog, 2008-7-9)

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.

<p>Esempio d'uso:</p> <pre> <b>final</b> MutexWithLog m = <b>new</b> MutexWithLog();  Thread t = <b>new</b> Thread("Secondo") {     <b>public void</b> run() {         m.lock();         System.out.println("Due!");         m.unlock();     } }; t.start(); m.lock(); System.out.println("Uno!"); m.unlock(); </pre>	<p>Un possibile output dell'esempio d'uso:</p> <pre> "main" ha acquisito il lock Uno! "main" ha rilasciato il lock "Secondo" ha acquisito il lock Due! "Secondo" ha rilasciato il lock </pre>
---	---

## 57. (RunnableWithProgress, 2008-6-19)

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.

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

## 58. (DelayIterator, 2008-3-27)

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

```

## 59. (Simulazione di ParkingLot, 2007-7-20)

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.

60. (Highway, 2007-6-29)

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>

61. (2006-9-15)

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 }
```



## 16 Iteratori e ciclo for-each

### 1. (CrazyIterator, 2014-11-3)

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 }
```

### 2. (IncreasingSubsequence, 2009-9-18)

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&lt;Integer&gt; l = new LinkedList&lt;Integer&gt;(); l.add(10); l.add(3); l.add(5); l.add(12); l.add(11); l.add(35); for (Integer i: new IncreasingSubseq&lt;Integer&gt;(l))     System.out.println(i);</pre>	<pre>10 12 35</pre>

### 3. (CrazyIterator, 2008-4-21)

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 }

```

## 4. (MyFor, 2008-2-25)

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) { ... }
```

## 5. (Selector, 2007-9-17)

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&lt;Integer&gt; l = Arrays.asList(a); Selector&lt;Integer&gt; pari = new Selector&lt;Integer&gt;() {     public boolean select(Integer n) {         return (n % 2) == 0;     } }; for (Integer n: new SelectorIterator&lt;Integer&gt;(l, pari))     System.out.print(n + " "); </pre>	<pre> 2 56 22 12 56 </pre>

## 6. (CommonDividers, 2007-7-20)

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>	1 2 3 4 6 12

#### 7. (AncestorIterator, 2007-4-26)

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>	Luca Rossi Mario Rossi Luisa Verdi

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**.

#### 8. (Primes, 2007-2-23)

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 &gt; 20) break;     System.out.println(i); }</pre>	1 3 5 7 11 13 17 19

#### 9. (SuperclassIterator, 2006-9-15)

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.

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

10. (**TwoSteps, 2006-7-17**)

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&lt;Integer&gt; l = new LinkedList&lt;Integer&gt;(); l.add(3); l.add(5); l.add(7); l.add(9);  Iterator&lt;Integer&gt; iter1 = twoSteps(l.iterator()); System.out.println("Iterazione_1:"); System.out.println(iter1.next()); System.out.println(iter1.next());  Iterator&lt;Integer&gt; 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>

11. (**BinaryTreePreIterator, 2006-4-27**)

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();
        ...
    }
}
```



# 17 Criteri di ordinamento tra oggetti

## 1. (Rotating list comparator, 2019-4-29)

Dire quali dei seguenti sono criteri validi per un comparatore tra oggetti RotatingList dell'esercizio precedente. In caso di validità, dire anche se il criterio è coerente con la definizione di equals dell'esercizio precedente.

compare(x, y) restituisce (nei casi non contemplati, il metodo restituisce 0):

- a) -1 se le due liste hanno lunghezza almeno 3 e x si può trasformare in y tramite una singola rotazione a destra; 1 se le due liste hanno lunghezza almeno 3 e x si può trasformare in y tramite una singola rotazione a sinistra.
- b) -1 se x è un prefisso proprio di y (come "casa" è un prefisso proprio di "casata"); 1 se y è un prefisso proprio di x.
- c) -1 se y contiene un oggetto che non è presente in x; 1 se x contiene tutti gli oggetti di y, e anche un oggetto non presente in y.
- d) -1 se x ha lunghezza pari e y dispari; 1 se y ha lunghezza pari e x dispari.

## 2. (Rotating list comparator, 2019-4-29)

Data la seguente interfaccia funzionale:

```
public interface Predicate<T> {  
    boolean test(T t);  
}
```

usare una lambda-espressione per definire ciascuna delle seguenti implementazioni dell'interfaccia:

- a) Un predicato che accetta un intero e restituisce true se è pari.
- b) Un predicato che accetta una stringa e restituisce true se quella stringa è uguale a "exit".
- c) Un predicato che accetta un SortedSet<Integer> e restituisce true se l'elemento minimo dell'insieme è negativo e quello massimo è positivo.
- d) Un predicato che accetta una Collection<Object> e restituisce true se la collezione contiene almeno un duplicato (secondo equals).

## 3. (Date, 2018-6-20)

La classe Date rappresenta una data tramite tre numeri interi (giorno, mese e anno) e ridefinisce equals nel modo naturale.

Dire quali delle seguenti sono specifiche valide per un comparatore tra due oggetti Date a e b. Dire anche quali specifiche sono coerenti con equals.

compare(a,b) restituisce (nei casi non contemplati, restituisce 0):

- a) -1 se l'anno di a è minore di quello di b; 1 se l'anno di a è maggiore di quello di b.
- b) -1 se a e b hanno lo stesso mese; 1 se a e b hanno mesi diversi.
- c) -1 se il mese di a è tra gennaio e giugno e quello di b tra luglio e dicembre; 1 se il mese di b è tra gennaio e giugno e quello di a tra luglio e dicembre.
- d) -1 se il giorno oppure il mese di a è uguale a quello di b; 1 se sia il giorno sia il mese di a sono diversi da quelli di b.

## 4. (Product, 2018-5-2)

[CROWDGRADER] Realizzare la classe Product, che rappresenta un prodotto di un supermercato, caratterizzato da descrizione e prezzo. I prodotti sono dotati di ordinamento naturale, in base

alla loro descrizione (ordine alfabetico). Il metodo `getMostExpensive` restituisce il prodotto più costoso. Il campo `comparatorByPrice` contiene un comparatore tra prodotti, che confronta i prezzi.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre>Product a = new Product("Sale", 0.60),       b = new Product("Zucchero", 0.95),       c = new Product("Caffe'", 2.54);  System.out.println(Product.getMostExpensive()); System.out.println(b.compareTo(c)); System.out.println(Product.comparatorByPrice.compare(b, c));</pre>	<pre>Caffe', 2.54 1 -1</pre>

#### 5. (Sphere Comparator, 2017-6-21)

La classe `Sphere` rappresenta una sfera nello spazio, caratterizzata dalle coordinate del centro e dal raggio. Due sfere sono uguali secondo `equals` se hanno lo stesso stesso raggio (indipendentemente dal centro). Dire quali delle seguenti sono specifiche valide per un comparatore tra due oggetti `Sphere` *a* e *b*. Dire anche quali specifiche sono coerenti con `equals`.

`compare(a,b)` restituisce (nei casi non contemplati, restituisce 0):

- a) -1 se il volume di *a* è minore di quello di *b*; 1 se il volume di *a* è maggiore di quello di *b*.
- b) -1 se *a* e *b* hanno lo stesso centro; 1 se *a* e *b* hanno centri diversi.
- c) -1 se la somma dei due raggi è minore della distanza tra i centri; 1 se la somma dei due raggi è maggiore della distanza tra i centri.
- d) -1 se i raggi sono diversi; 1 se i raggi sono uguali, ma i centri sono diversi.

#### 6. (Book, 2016-7-21)

Implementare la classe **Book**, che rappresenta un libro diviso in capitoli. Il metodo **addChapter** aggiunge un capitolo in coda al libro, caratterizzato da titolo e contenuto. I capitoli sono automaticamente numerati a partire da 1. Il metodo **getChapterName(i)** restituisce il titolo del capitolo *i*-esimo, mentre il metodo **getChapterContent(i)** ne restituisce il contenuto.

Gli oggetti **Book** devono essere clonabili. Inoltre, la classe deve essere dotata di ordinamento naturale, basato sul numero di capitoli.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:	Output:
<pre>Book b = new Book(); b.addChapter("Prefazione", "Sono_passati_pochi_anni..."); b.addChapter("Introduzione", "Un_calcolatore_digitale..."); ; b.addChapter("Sistemi_di_elaborazione", "Un_calcolatore..."); Book bb = b.clone(); System.out.println(bb.getChapterContent(1)); System.out.println(bb.getChapterTitle(2));</pre>	<pre>Sono passati pochi anni... Introduzione</pre>

#### 7. (Set of Integer comparator, 2016-6-22)

Dire quali delle seguenti sono specifiche valide per un **Comparator** tra due oggetti di tipo **Set<Integer>**, motivando la risposta. Nei casi non previsti dalle specifiche, il comparatore restituisce 0.

**compare(a, b)** restituisce:

- a)  $-1$  se *a* contiene un numero minore di tutti i numeri di *b*;  $1$  se *b* contiene un numero minore di tutti i numeri di *a*
- b)  $-1$  se la somma dei numeri di *a* è minore della somma dei numeri di *b*;  $1$  se la media dei numeri di *b* è maggiore della media dei numeri di *a*
- c)  $-1$  se *a* contiene tutti numeri negativi e *b* no;  $1$  se *b* contiene tutti numeri positivi e *a* no
- d)  $-1$  se *a* contiene il numero 0;  $1$  se *a* non contiene il numero 0
- e)  $-1$  se *a* contiene il numero 0 e *b* no;  $1$  se *b* contiene il numero 0 e *a* no

#### 8. (Engine Comparator, 2016-4-21)

Con riferimento alla classe dell'esercizio 2, dire quali delle seguenti sono specifiche valide per un **Comparator** tra due oggetti di tipo **Engine**, motivando la risposta. Nei casi non previsti dalle specifiche, il comparatore restituisce 0.

**compare(a, b)** restituisce:

- a)  $-1$  se *a* ha cilindrata minore di *b*;  $1$  se *a* ha cilindrata maggiore di *b*
- b)  $-1$  se *a* ha potenza minore della metà di *b*;  $1$  se *a* ha potenza maggiore del doppio di *b*
- c)  $-1$  se *a* ha il rapporto potenza/cilindrata minore di *b*;  $1$  se *a* ha il rapporto potenza/cilindrata maggiore di *b*
- d)  $-1$  se *a* ha cilindrata oppure potenza minore di *b*;  $1$  se *a* ha sia cilindrata sia potenza maggiori di *b*
- e)  $-1$  se *a* ha cilindrata 1200 e potenza minore di *b*;  $1$  se *a* ha cilindrata 1200 e potenza maggiore di *b*

#### 9. (SetComparator, 2015-7-8)

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.

## 10. (Box, 2015-2-5)

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>

## 11. (DataSeries, 2015-1-20)

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:	Output:
<pre>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&lt;DataSeries&gt; c2005 = DataSeries.comparatorByYear(2005),   c2010 = DataSeries.comparatorByYear(2010); System.out.println(c2005.compare(pop1, pop2)); System.out.println(c2010.compare(pop1, pop2));</pre>	<pre>1 -1</pre>

## 12. (EmployeeComparator, 2014-9-18)

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.

### 13. (Playlist, 2014-7-28)

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>

### 14. (Pizza, 2014-11-3)

[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>	<pre> 1 </pre>

### 15. (String comparator, 2013-6-25)

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”.

### 16. (MaxBox, 2013-1-22)

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.

<p>Esempio d'uso:</p> <pre> MaxBox&lt;String&gt; bb1 = new MaxBox&lt;String&gt;(); MaxBox&lt;String&gt; bb2 = new MaxBox&lt;String&gt;(     new Comparator&lt;String&gt;() {         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>	<p>Output:</p> <pre> sette dodici </pre>
--	--

## 17. (Point, 2012-6-18)

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);`

## 18. (Time, 2010-9-14)

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`.

<p>Esempio d'uso:</p> <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>	<p>Output dell'esempio d'uso:</p> <pre> 7:24:30 1 -1 </pre>
--	---

## 19. (PartiallyComparable, 2010-6-28)

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&lt;POString&gt; { ... }  POSet&lt;POString&gt; set = new POSet&lt;POString&gt;(); 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>
---	---

## 20. (Rebus, 2010-5-3)

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)

## 21. (Version, 2010-2-24)

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>

## 22. (2010-11-30)

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:

- -1, se  $a$  è più corto di  $b$ ;
  - 1, se  $b$  è più corto di  $a$ ;
  - 0, altrimenti.
- -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
- -1, se  $a$  è orizzontale e  $b$  è verticale;
  - 1, se  $b$  è orizzontale e  $a$  è verticale;
  - 0, altrimenti.

## 23. (IncreasingSubsequence, 2009-9-1’8)

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&lt;Integer&gt; l = new LinkedList&lt;Integer&gt;(); l.add(10); l.add(3); l.add(5); l.add(12); l.add(11); l.add(35); for (Integer i: new IncreasingSubseq&lt;Integer&gt;(l))     System.out.println(i);</pre>	<pre>10 12 35</pre>

#### 24. (Circle, 2009-4-23)

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>

#### 25. (Triangle 2, 2009-11-27)

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&lt;Triangle&gt; pc = Triangle.     perimeterComparator(); System.out.println(pc.compare(b, c));</pre>	<pre>true true true -1</pre>

#### 26. (2009-1-15)

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&lt;Integer&gt; l = new LinkedList&lt;Integer&gt;();         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>

27. (Sorter, 2008-1-30)

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:
<pre> Sorter&lt;Integer&gt; s = new Sorter&lt;Integer&gt;(7);  System.out.println(s.check(4)); System.out.println(s.check(1)); System.out.println(s.check(6)); System.out.println(s.check(6)); </pre>	<pre> -1 -1 1 0 </pre>

28. (FunnyOrder, 2007-9-17)

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 + " ");
    }
}

```

29. (**Rational**, 2007-6-29)

- (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:	Output dell'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>	<pre> 1/6 13/30 2/45 </pre>



# 18 Indice Cronologico

- **2006-4-27:** Average, pag. 67 ; Esercizio 195, pag. 52 ; BinaryTreePreIterator, pag. 196 ; Esercizio 197, pag. 107 ;
- **2006-6-26:** Moto bidimensionale, pag. 67 ; Esercizio 189, pag. 51 ; Publication, pag. 92 ; Esercizio 191, pag. 161 ; DoubleQueue, pag. 92 ; Esercizio 193, pag. 93 107 120 ;
- **2006-7-17:** Moto accelerato, pag. 67 ; Esercizio 165, pag. 51 ; Spartito, pag. 91 ; Esercizio 167, pag. 160 ; TwoSteps, pag. 196 ; Esercizio 169, pag. 91 106 120 ;
- **2006-9-15:** FallingBody, pag. 66 ; Esercizio 171, pag. 50 ; TreeType, pag. 66 164 ; Esercizio 173, pag. 160 ; SuperclassIterator, pag. 168 195 ; Esercizio 175, pag. 106 191 ;
- **2007-1-12:** Polinomio, pag. 65 ; Esercizio 185, pag. 49 ; Esercizio 186, pag. 160 ; Insieme di polinomi, pag. 91 ;
- **2007-2-23:** Inventory, pag. 90 119 ; Primes, pag. 195 ; Esercizio 182, pag. 49 ; Esercizio 183, pag. 159 ;
- **2007-2-7:** Polinomio bis, pag. 90 119 ; Monomio, pag. 7 ; Esercizio 178, pag. 48 ; Esercizio 179, pag. 159 ;
- **2007-4-26:** Genealogia, pag. 64 ; AncestorIterator, pag. 195 ; Esercizio 41, pag. 48 ; Esercizio 42, pag. 65 105 ;
- **2007-6-29:** Rational, pag. 64 207 ; Esercizio 54, pag. 47 ; Polinomio su un campo generico, pag. 118 ; Esercizio 56, pag. 89 105 ; Esercizio 57, pag. 159 ; Highway, pag. 89 191 ;
- **2007-7-20:** CommonDividers, pag. 117 194 ; Esercizio 34, pag. 47 ; ParkingLot, pag. 63 117 ; Esercizio 36, pag. 105 118 ; Esercizio 37, pag. 158 ; Simulazione di ParkingLot, pag. 190 ;
- **2007-9-17:** Aereo, pag. 63 ; Esercizio 44, pag. 46 ; Selector, pag. 116 194 ; FunnyOrder, pag. 88 117 206 ; Esercizio 47, pag. 158 ;
- **2008-1-30:** Recipe, pag. 88 ; Esercizio 65, pag. 46 ; Sorter, pag. 116 206 ; Esercizio 67, pag. 63 126 ; Esercizio 68, pag. 157 ;
- **2008-2-25:** BoolExpr, pag. 87 115 ; Esercizio 60, pag. 45 ; MyFor, pag. 116 194 ; Esercizio 62, pag. 63 126 ; Esercizio 63, pag. 157 ;
- **2008-3-27:** Impianto e Apparecchio, pag. 62 87 ; Esercizio 49, pag. 45 ; DelayIterator, pag. 190 ; Esercizio 51, pag. 125 ; Esercizio 52, pag. 157 ;
- **2008-4-21:** Triangolo, pag. 62 132 ; Esercizio 251, pag. 44 ; Esercizio 252, pag. 7 ; CrazyIterator, pag. 193 ;
- **2008-6-19:** Molecola, pag. 87 ; Esercizio 237, pag. 44 ; RunnableWithProgress, pag. 190 ; Esercizio 239, pag. 125 ; Esercizio 240, pag. 156 ;
- **2008-7-9:** Esercizio 241, pag. 7 ; Esercizio 242, pag. 43 ; MutexWithLog, pag. 189 ; Esercizio 244, pag. 125 ; Esercizio 245, pag. 156 ;
- **2008-9-8:** PostIt, pag. 86 ; Esercizio 260, pag. 43 ; RunnableWithArg, pag. 189 ; Esercizio 262, pag. 62 ; Esercizio 263, pag. 156 ;
- **2009-1-15:** Anagramma, pag. 61 ; Esercizio 232, pag. 42 ; Volo e Passeggero, pag. 86 ; Esercizio 234, pag. 124 205 ; Esercizio 235, pag. 155 ;

- **2009-1-29:** Interval, pag. 61 132 ; Esercizio 247, pag. 42 ; Split, pag. 115 ; Esercizio 249, pag. 155 ;
- **2009-11-27:** Triangle 2, pag. 205 ; Esercizio 204, pag. 41 ; CountByType, pag. 86 167 ; Esercizio 206, pag. 124 131 ; Esercizio 207, pag. 154 ;
- **2009-2-19:** Container, pag. 85 ; Esercizio 255, pag. 41 ; Interleave, pag. 115 ; Esercizio 257, pag. 124 ; Esercizio 258, pag. 154 ;
- **2009-4-23:** UML, pag. 85 ; Esercizio 223, pag. 40 ; Circle, pag. 61 205 ; Esercizio 225, pag. 123 131 ;
- **2009-6-19:** Tutor, pag. 84 ; Esercizio 199, pag. 40 ; Cardinal, pag. 135 ; Esercizio 201, pag. 115 123 ; Esercizio 202, pag. 154 ;
- **2009-7-9:** Washer, pag. 130 ; Esercizio 209, pag. 39 ; Elevator, pag. 189 ; Esercizio 211, pag. 115 123 ; Esercizio 212, pag. 153 ;
- **2009-9-1'8:** Auction, pag. 188 ; Esercizio 227, pag. 39 ; IncreasingSubsequence, pag. 193 204 ; Esercizio 229, pag. 123 130 ; Esercizio 230, pag. 153 ;
- **2010-1-22:** Color, pag. 60 84 ; Esercizio 214, pag. 38 ; GetByType, pag. 84 167 ; Esercizio 216, pag. 122 130 164 ; Esercizio 217, pag. 153 ;
- **2010-11-30:** Segment, pag. 60 164 ; Esercizio 29, pag. 38 ; Esercizio 30, pag. 204 ; SelectKeys, pag. 84 114 ; Esercizio 32, pag. 152 ;
- **2010-2-24:** Wall, pag. 60 ; Esercizio 219, pag. 37 ; Version, pag. 204 ; Esercizio 221, pag. 152 ;
- **2010-5-3:** Crosswords, pag. 59 ; Esercizio 7, pag. 37 ; Rebus, pag. 203 ;
- **2010-6-28:** QueueOfTasks, pag. 188 ; Esercizio 10, pag. 36 ; PartiallyComparable, pag. 83 202 ; Esercizio 12, pag. 151 ;
- **2010-7-26:** Tetris, pag. 58 ; Esercizio 2, pag. 36 ; TetrisPiece, pag. 135 ; Esercizio 4, pag. 114 122 ; Esercizio 5, pag. 151 ;
- **2010-9-14:** Time, pag. 58 202 ; Esercizio 24, pag. 35 ; Intersect, pag. 83 114 ; ExecuteInParallel, pag. 188 ; Esercizio 27, pag. 151 ;
- **2011-2-7:** VoteBox, pag. 187 ; Esercizio 14, pag. 35 ; Operai, pag. 6 ; MakeMap, pag. 83 114 ; Esercizio 17, pag. 150 ;
- **2011-3-4:** MultiProgressBar, pag. 187 ; Esercizio 19, pag. 34 ; Esercizio 20, pag. 114 122 ; PrintBytes, pag. 58 ; Esercizio 22, pag. 150 ;
- **2012-4-23:** Esercizio 264, pag. 34 ; Safe, pag. 57 ; Esercizio 266, pag. 6 ; Panino, pag. 82 135 ;
- **2012-6-18:** Esercizio 273, pag. 33 ; Point, pag. 202 ; BoundedMap, pag. 82 113 ; ThreadRace, pag. 187 ; Esercizio 277, pag. 150 ;
- **2012-7-9:** Mystery thread, pag. 186 ; Esercizio 269, pag. 32 ; Social network, pag. 82 ; NumberType, pag. 135 ; Esercizio 272, pag. 149 ;
- **2012-9-3:** Esercizio 287, pag. 32 ; Mystery thread 2, pag. 186 ; Anagrammi, pag. 6 163 ; Bijection, pag. 81 ; Esercizio 291, pag. 149 ;
- **2013-1-22:** Esercizio 292, pag. 32 ; Shared object, pag. 185 ; Insieme di lettere, pag. 5 163 ; MaxBox, pag. 201 ; Esercizio 296, pag. 149 ;
- **2013-12-16:** Esercizio 154, pag. 31 ; Note, pag. 134 ; concurrentMax, pag. 185 ; agree, pag. 103 ; Esercizio 158, pag. 148 ;
- **2013-2-11:** Esercizio 278, pag. 30 ; MultiSet, pag. 5 81 ; Concurrent filter, pag. 185 ; Esercizio 281, pag. 148 ;

- **2013-3-22:** Esercizio 282, pag. 30 ; Auditorium, pag. 81 ; Cane, pag. 5 ; Shared average, pag. 184 ; Esercizio 286, pag. 148 ;
- **2013-4-29:** Esercizio 130, pag. 29 ; City, pag. 80 ; Pair, pag. 113 ; Esercizio 133, pag. 5 ;
- **2013-6-25:** Esercizio 134, pag. 29 ; String comparator, pag. 201 ; MultiBuffer, pag. 80 184 ; Concat, pag. 80 102 ; Esercizio 138, pag. 148 ;
- **2013-7-9:** Esercizio 159, pag. 28 ; BloodType, pag. 134 ; processArray, pag. 183 ; isSorted, pag. 79 102 ; Esercizio 163, pag. 147 ;
- **2013-9-25:** Esercizio 139, pag. 28 ; Movie, pag. 79 ; executeWithDeadline, pag. 183 ; composeMaps, pag. 79 102 ; Esercizio 143, pag. 147 ;
- **2014-1-31:** Esercizio 149, pag. 27 ; BoundedSet, pag. 78 ; PostOfficeQueue, pag. 183 ; isMax, pag. 101 ; Esercizio 153, pag. 147 ;
- **2014-11-28:** Esercizio 82, pag. 27 ; Coin, pag. 134 ; Alarm, pag. 183 ; product, pag. 101 ; Esercizio 86, pag. 147 ;
- **2014-11-3:** Esercizio 102, pag. 26 ; Pizza, pag. 130 134 201 ; CrazyIterator, pag. 193 ;
- **2014-3-5:** Esercizio 144, pag. 26 ; PeriodicTask, pag. 4 182 ; Status, pag. 133 ; extractPos, pag. 101 ; Esercizio 148, pag. 146 ;
- **2014-4-28:** Esercizio 69, pag. 25 ; Shape, pag. 3 ; Shape equals, pag. 4 ;
- **2014-7-28:** Esercizio 77, pag. 25 ; Playlist, pag. 57 201 ; PriorityExecutor, pag. 182 ; inverseMap, pag. 78 101 ; Esercizio 81, pag. 146 ;
- **2014-7-3:** Esercizio 72, pag. 24 ; NutrInfo, pag. 133 ; Exchanger, pag. 181 ; subMap, pag. 78 100 ; Esercizio 76, pag. 146 ;
- **2014-9-18:** Esercizio 87, pag. 24 ; EmployeeComparator, pag. 200 ; Contest, pag. 77 ; atLeastOne, pag. 181 ; Esercizio 91, pag. 146 ;
- **2015-1-20:** Esercizio 92, pag. 23 ; DataSeries, pag. 200 ; Relation, pag. 77 113 ; difference, pag. 100 ; Esercizio 96, pag. 145 ;
- **2015-2-5:** Esercizio 110, pag. 23 ; Box, pag. 57 200 ; ForgetfulSet, pag. 181 ; reverseList, pag. 100 ; Esercizio 114, pag. 145 ;
- **2015-6-24:** Esercizio 105, pag. 22 ; SimpleThread, pag. 181 ; Controller, pag. 76 129 ; listIntersection, pag. 100 ; Esercizio 109, pag. 145 ;
- **2015-7-8:** Esercizio 97, pag. 22 ; TimeToFinish, pag. 180 ; Question e Answer, pag. 56 ; SetComparator, pag. 199 ; Esercizio 101, pag. 144 ;
- **2015-9-21:** Esercizio 115, pag. 21 ; Progression, pag. 76 ; StringQuiz, pag. 180 ; splitList, pag. 99 ; Esercizio 119, pag. 144 ;
- **2016-1-27:** Esercizio 120, pag. 21 ; Curriculum, pag. 75 129 ; twoPhases, pag. 180 ; Esercizio 123, pag. 112 121 ; Esercizio 124, pag. 144 ;
- **2016-3-3:** Esercizio 125, pag. 20 ; GameLevel, pag. 56 75 ; MysteryThread3, pag. 179 ; Soldier, pag. 3 ; Esercizio 129, pag. 144 ;
- **2016-4-21:** Esercizio 297, pag. 20 ; Engine, pag. 3 128 ; Engine Comparator, pag. 199 ; Count, pag. 109 ;
- **2016-6-22:** Esercizio 301, pag. 19 ; Set of Integer comparator, pag. 199 ; BlockingArray, pag. 179 ; arePermutations, pag. 99 ; Esercizio 305, pag. 143 ;
- **2016-7-21:** Esercizio 306, pag. 19 ; Book, pag. 56 163 199 ; findString, pag. 179 ; Esercizio 309, pag. 112 121 ; Esercizio 310, pag. 143 ;

- **2016-9-20:** Esercizio 311, pag. 18 ; SocialUser, pag. 75 ; Somma due, pag. 178 ; Esercizio 314, pag. 143 ;
- **2017-1-25:** Esercizio 315, pag. 18 ; LengthUnit, pag. 133 ; mergeIfSorted, pag. 75 178 ; Esercizio 318, pag. 142 ;
- **2017-10-6:** Esercizio 341, pag. 17 ; Clinica, pag. 74 ; Somma e azzera, pag. 178 ; Esercizio 344, pag. 142 ;
- **2017-2-23:** Esercizio 319, pag. 17 ; Polygon, pag. 2 ; sumAndMax, pag. 178 ; Esercizio 322, pag. 142 ;
- **2017-3-23:** Esercizio 323, pag. 16 ; BinRel, pag. 74 ; Bonus per Employee, pag. 177 ; Esercizio 326, pag. 141 ;
- **2017-4-26:** Esercizio 327, pag. 16 ; Room, pag. 73 ; Room equals, pag. 2 ; Container, pag. 112 ;
- **2017-6-21:** Esercizio 331, pag. 15 ; Sphere Comparator, pag. 198 ; Market, pag. 177 ; findNext, pag. 99 ; Esercizio 335, pag. 141 ;
- **2017-7-20:** Esercizio 336, pag. 15 ; Cartella, pag. 55 ; MysteryThread4, pag. 176 ; common-Keys, pag. 99 ; Esercizio 340, pag. 141 ;
- **2018-1-24:** Bug, pag. 73 ; Shared total, pag. 176 ; isIncreasing, pag. 98 ; Esercizio 348, pag. 141 ;
- **2018-10-18:** Esercizio 374, pag. 14 ; Component e Configuration, pag. 72 ; greatestLowerBound, pag. 98 ; Esercizio 377, pag. 140 ;
- **2018-2-22:** Book e Library, pag. 72 ; Two threads, pag. 175 ; cartesianProduct, pag. 98 ; Esercizio 356, pag. 140 ;
- **2018-3-23:** Esercizio 349, pag. 14 ; Studente, pag. 2 128 ; isSetSmaller, pag. 72 ; Esercizio 352, pag. 140 ;
- **2018-5-2:** Esercizio 357, pag. 13 ; Product, pag. 197 ; Merge, pag. 71 109 ;
- **2018-6-20:** Esercizio 365, pag. 12 ; Date, pag. 197 ; PeriodicExecutor, pag. 175 ; makeMap, pag. 97 ; Esercizio 369, pag. 139 ;
- **2018-7-19:** Esercizio 360, pag. 12 ; Fraction, pag. 1 ; SafeSet, pag. 71 174 ; MysteryThread5, pag. 175 ; Esercizio 364, pag. 139 ;
- **2018-9-17:** Esercizio 370, pag. 12 ; Cellphone, pag. 70 ; SharedCounter, pag. 174 ; Esercizio 373, pag. 139 ;
- **2019-1-23:** Esercizio 378, pag. 11 ; GuessTheNumber, pag. 174 ; findPrevious, pag. 97 ; Esercizio 381, pag. 139 ;
- **2019-10-9:** Shop, pag. 173 ; Esercizio 409, pag. 121 ; interleave2, pag. 97 ; Esercizio 411, pag. 138 ;
- **2019-2-15:** Esercizio 382, pag. 11 ; Range, pag. 111 ; Missing synch 2, pag. 173 ; Esercizio 385, pag. 138 ;
- **2019-3-19:** Esercizio 386, pag. 10 ; Library, pag. 70 ; Missing synch 3, pag. 172 ; Esercizio 389, pag. 138 ;
- **2019-4-29:** Esercizio 390, pag. 10 ; RotatingList, pag. 69 ; Rotating list comparator, pag. 197 ; Rotating list comparator, pag. 197 ;
- **2019-6-24:** Esercizio 394, pag. 9 ; Box e ColoredBox, pag. 1 ; SortedList, pag. 69 ; keysWithValue, pag. 97 ; Esercizio 398, pag. 138 ;



- **2019-7-23:** Esercizio 399, pag. 9 ; Student, pag. 1 ; RandomExecutor, pag. 172 ; Minimum enum, pag. 96 ; Esercizio 403, pag. 137 ;
- **2019-9-20:** Microwave, pag. 127 ; disjoint, pag. 96 ; MysteryThread6, pag. 171 ; Esercizio 407, pag. 137 ;
- **2020-1-24:** Product e Cart, pag. 69 ; MysteryThread7, pag. 171 ; Uguaglianza tra cart, pag. 1 ; Esercizio 415, pag. 137 ;
- **2020-2-27:** Accumulator, pag. 111 ; MysteryThread8, pag. 170 ; keysWithHighestValue, pag. 96 ; Esercizio 419, pag. 137 ;
- **2021-10-26:** WiFi, pag. 54 ; Missing synch 3, pag. 170 ; Esercizio 430, pag. 167 ; countInBetween, pag. 96 ;
- **2021-7-26:** GreenPass, pag. 54 ; InternalLayout1, pag. 127 ; overridingMap, pag. 95 ; Esercizio 423, pag. 169 ;
- **2021-9-24:** Radio, pag. 53 ; InternalLayout2, pag. 127 ; MysteryThread6, pag. 169 ; countOccurrences, pag. 95 ;
- **2022-1-26:** Exchange, pag. 53 ; Esercizio 433, pag. 167 ; Esercizio 434, pag. 167 ; combine, pag. 95 ;