Eserciziario di Programmazione ad Oggetti mod. 2

a.a. 2021/2022

1 Tests

1.1 Design Patterns

Questa sezione contiene domande sull'argomento design patterns.

1. Si prenda in considerazione il seguente codice Java:

```
public class MyClass {
    private static MyClass instance;
    public static MyClass get() {
        if (instance == null) instance = new MyClass();
        return instance;
    }
}
```

   Quale design pattern è all'opera?

   ◯ Command ◯ Factory ◯ nessun pattern ◯ Singleton

2. Perché il design pattern denominato *factory* è utile in Java ed in altri linguaggi ad oggetti simili?

   ◯ Perché i costruttori non sono metodi soggetti al *dynamic dispatching*, quindi non è possibile sfruttare il polimorfismo per costruire oggetti se non tramite un metodo non-statico che incapsula la costruzione.

   ◯ Perché la costruzione degli oggetti va nascosta dentro metodi statici al fine di poter rendere privati i costruttori, applicando così i principi di *information hiding* e di incapsulamento tipici della programmazione ad oggetti anche alla creazione di istanze.

   ◯ Siccome le sottoclassi non hanno accesso ai super-costruttori, è necessario dotare le superclassi di metodi wrapper `protected` dei propri costruttori, permettendo alle classi derivate di accedervi indirettamente.

   ◯ Perché non è possibile costruire array con tipi generici in Java, pertanto è preferibile mascherare gli inevitabili cast dentro un metodo statico che funge da wrapper per il costruttore.

1.2 Riuso di codice e Type system di Java

Questa sezione contiene domande sull'argomento polimorfismo, genericità e typing, compresi i temi di *subtyping*, *generics*, *override*, *overloading*, ecc.

1. Java versione 10 supporta l'ereditarietà multipla?

   ◯ Sì, ma solo tra classi, mentre è possibile implementare solamente una interfaccia alla volta.

   ◯ No, nemmeno tra interfacce, a causa del noto *problema del diamante*.

   ◯ Sì, perché risolve il *problema del diamante* mescolando opportunamente le *virtual table* delle superclassi secondo l'ordine di apparizione di queste ultime dopo la keyword `extends`.

   ◯ No, però non c'è limite al numero di interfacce che una classe può implementare.

2. In generale, a cosa servono le due forme di polimorfismo (subtyping e generics) che offre Java oggi?

   ◯ A riusare lo stesso codice, ovvero chiamare un metodo, con parametri di tipo differente.

   ◯ A definire classi che implementano due o più interfacce.

   ◯ A riusare lo stesso codice, in termini di classi e dei loro membri, ereditandoli anziché riscrivendoli.

   ◯ A permettere di definire classi o interfacce parametriche su altri tipi.

3. Si assuma la relazione di subtyping $\preceq$ (ed il suo complemento $\not\preceq$) tale per cui, dati due tipi A e B, se $A \preceq B$ allora A è *sottotipo* di B. E siano $\tau, \sigma, \rho, \varphi$ tipi concreti per cui valgono le seguenti relazioni: $\sigma \preceq \tau$, $\rho \preceq \sigma$ e $\varphi \preceq \tau$. Si indichi quali delle seguenti relazioni *non* è valida.

   ◯ $\rho \preceq \tau$

   ◯ $\sigma \not\preceq \rho$

   ◯ $\varphi \preceq \rho$

   ◯ $\tau \not\preceq \varphi$

4. L'ereditarietà è una forma di polimorfismo?
   - ◯ Sì, perché permette il riuso del codice di una classe e dei suoi membri pubblici anche nelle sottoclassi.
   - ◯ No, ma rende possibile il polimorfismo subtype poiché garantisce che i membri pubblici delle superclassi esistano nelle sottoclassi.
   - ◯ Si, perché, grazie al dynamic dispatching, viene garantita l'invocazione corretta a run-time dei metodi in override nelle sottoclassi anche quando i tipi a compile-time sono soggetti a *subsumption*.
   - ◯ No, perché l'ereditarietà non garantisce davvero che i membri pubblici delle superclassi esistano anche nelle sottoclassi, in quanto è possibile cambiare la visibilità dei membri pubblici ereditati al momento dell'override, ad esempio rendendoli privati, quindi non invocabili.

5. Quali parti della firma di un metodo sono coinvolte nella risoluzione dell'overloading in Java?
   - ◯ Il tipo ed il numero dei parametri, il tipo di ritorno ma non le eccezioni dichiarate.
   - ◯ Il tipo ed il numero dei parametri, il tipo di ritorno ed anche le eccezioni dichiarate.
   - ◯ Il tipo ed il numero dei parametri e basta, senza tipo di ritorno né eccezioni.
   - ◯ Il tipo ed il numero dei parametri inizialmente; se sono ambigui anche il tipo di ritorno, ma mai le eccezioni.

6. Si prenda in considerazione il seguente codice Java 8+ contenente una semplice gerarchia di classi ed un metodo generico di nome `map`:

```
public class Rpg {
        public static <A, B> Collection<B> map(Collection<A> c, Function<A, B> f) {
                List<B> r = new ArrayList<>();
                for (A a : c) {
                        r.add(f.apply(a));
                }
                return r;
        }

        public static abstract class Character<R extends Number> {
                public int level;
                public final String name;
                protected Character(int level, String name) {
                        this.level = level;
                        this.name = name;
                }
                public abstract R attack();
        }

        public static class Paladin extends Character<Float> {
                public float mana;
                public Paladin(int level, String name) {
                        super(level, name);
                        mana = 100.f;
                }
                @Override
                public Float attack() { return mana * level / 2.f; }
        }

        public static class Rogue extends Character<Number> {
                public int energy;
                public Rogue(int level, String name) {
                        super(level, name);
                        energy = 50;
                }
                @Override
```

```
                public Integer attack() { return (energy -= 35) > 20 ? level * 2 : 0; }
        }
}
```

Gli *override* del metodo `attack()` sono validi oppure no?

- ◯ No, perché le sottoclassi `Paladin` e `Rogue`, specializzando il tipo di ritorno rispetto al type argument passato alla superclasse, violano la regola della contro-varianza del tipo di ritorno che Java supporta.
- ◯ Sì, anche se le sottoclassi `Paladin` e `Rogue` specializzano il tipo di ritorno rispetto al type argument passato alla superclasse, la specializzazione del tipo di ritorno è supportata dall'overriding in Java.
- ◯ Non quello della classe `Rogue`, poiché passa `Number` come type argument alla superclasse, ma dichiara `Integer` (che è un sottotipo) come tipo di ritorno dell'override.
- ◯ Sì, perché la *type erasure* converte il generic `R` nel suo constraint `Number` ed i tipi di ritorno si possono specializzare rispetto ad esso, seguendo la regola della controvarianza.

7. In relazione al codice della domanda 6 (Sezione 1.2), si consideri ora il seguente codice *main*:

```
List<Paladin> retadins = new ArrayList<>();
retadins.add(new Paladin(60, "Leeroy Jenkins"));
retadins.add(new Paladin(80, "Arthas"));
```

Affinché il seguente statement compili, quali modifiche alla firma della funzione `map()` sarebbero necessarie e sufficienti?

```
Collection<Number> r1 = map(retadins, new Function<Character, Float>() {
        @Override
        public Float apply(Character c) {
                return 1.3f * (float) c.attack();
        }
});
```

- ◯ `static <A, B> Collection<B> map(Collection<A> c, Function<? super A, ? extends B> f)`
- ◯ `static <A, B> Collection<B> map(Collection<A> c, Function<? extends A, ? super B> f)`
- ◯ `static <A, B> Collection<B> map(Collection<? super A> c, Function<A, B> f)`
- ◯ `static <A, B> Collection<B> map(Collection<? extends A> c, Function<A, ? extends B> f)`

8. In relazione al codice della domanda 6 (Sezione 1.2), si prendano ora in considerazione i due seguenti metodi in overload:

```
public static int normalizeAttack(Character c) { return 1 + (int) c.attack(); }

public static Float normalizeAttack(Paladin c) { return c.attack(); }
```

(a) Sarebbero validi se aggiunti come membri alla classe `Rpg`?

- ◯ Sì, perché il tipo di ritorno discrimina univocamente la risoluzione dell'overloading per il compilatore.
- ◯ No, perché i metodi statici non supportano l'overloading.
- ◯ Sì, perché i due metodi hanno parametri di tipo differente; e nonostante `Character` sia un supertipo di `Paladin` il compilatore è in grado di discriminare.
- ◯ No, perché sebbene i due metodi abbiano parametri di tipo differente, `Character` è un supertipo di `Paladin`, quindi il compilatore non sarebbe in grado di risolverre l'overloading senza ambiguità.

(b) Il seguente statement compilerebbe, assumendo di conservare la firma originale della funzione `map()`? E se sì, quale overload di `normalizeAttack()` verrebbe passato come *method reference*?

```
Collection<Number> r2 = map(retadins, Rpg::normalizeAttack);
```

- ◯ Sì, compilerebbe e risolverebbe `normalizeAttack(Paladin)`.
- ◯ Sì, compilerebbe ed risolverebbe `normalizeAttack(Character)`.
- ◯ No, non compilerebbe senza modificare la firma della `map()`.
- ◯ No, non compilerebbe comunque perché nessuno dei due overload ha tipo di ritorno `Number`.

9. Si prendano in considerazione le seguenti interfacce Java:

```
public interface Solid extends Comparable<Solid> {
        double area();
        double volume();
        PositionedSolid at(Point origin);

        default int compareTo(Solid s) {
                Function<S, Double> f = (x) -> x.volume();
                return Double.compare(f.apply(this), f.apply(s))
        }
}

public interface Polyhedron extends Solid {
        double perimeter();
        @Override
        PositionedPolyhedron at(Point origin);
}

public interface PositionedSolid {
        Point origin();
}

public interface PositionedPolyhedron extends PositionedSolid, Iterable<Point> {}
```

Quale caratteristica del linguaggio Java è all'opera nell'override del metodo `at()` all'interno dell'interfaccia `Polyhedron`?

- ◯ E' un overload, non un override.
- ◯ L'overriding supporta la *co-varianza* del tipo di ritorno di un metodo.
- ◯ L'overriding supporta la *contro-varianza* del tipo di ritorno di un metodo.
- ◯ Un override può cambiare liberamente il tipo di ritorno di un metodo.

10. Si prenda in consederazione il seguentre codice:

```
public class Random {
    public Random() { ... }
    public Random(int seed) { ... }
    public boolean nextBoolean() { ... }
    public int nextInt() { ... }
    public double nextDouble() { ... }
}
```

I metodi che essa presenta hanno nomi diversi: sarebbe stato possibile definirli tutti con lo stesso nome (ad esempio `next()`) mantenendo inalterate le firme e la semantica?

- ◯ Sì: l'overloading sarebbe possibile e permetterebbe anche l'invocazione polimorfa di `next()` con risoluzione dipendente dal contesto;
- ◯ Non è necessario: Java offre una forma speciale di risoluzione dipendente dal contesto per metodi aventi nome simile e firma differente solo per il tipo di ritorno;
- ◯ No: l'overloading non è possibile tra metodi che differiscono solamente per il tipo di ritorno;
- ◯ Sì: l'overloading sarebbe possibile, tuttavia non darebbe alcun beneficio pratico poiché Java ad oggi non consente la risoluzione dell'overloading dipendente dal contesto.

1.3 Specificità e funzionalità Java

Questa sezione contiene domande su interfacce, metodi, costrutti esistenti nelle Java API e funzionalità specifiche dalle release Java, come ad esempio programmazione ibrida funzionale, ecc.

1.3.1 Equals

1. Si prenda in considerazione il seguente codice Java:

```
    public interface Equatable<T> {
        boolean equalsTo(T x);
}

public static class Person<P extends Person<P>> implements Equatable<P> {

        public final String name;
        public final int age;

        public Person(String name, int age) {
                this.name = name;
                this.age = age;
        }

        @Override
        public boolean equals(Object o) {
                if (this == o)
                        return true;
                if (o == null)
                        return false;
                if (getClass() != o.getClass())
                        return false;

                return equalsTo((P) o);
        }

        @Override
        public boolean equalsTo(P other) {
                return name.equals(other.name) && age == other.age;
        }

        @Override
        public String toString() {
                return name;
        }

}

public class Artist extends Person<Artist> {
        public final Hair hair;

        public Artist(String name, int age, Hair hair) {
                super(name, age);
                this.hair = hair;
        }

        @Override
        public boolean equalsTo(Artist other) {
                return super.equalsTo(other) && hair.equals(other.hair);
        }
}

public class Hair implements Equatable<Hair> {
```

```
        public final int length;
        public final Set<Color> colors;

        public Hair(int length, Set<Color> colors) {
                this.colors = colors;
                this.length = length;
        }

        @Override
        public boolean equals(Object o) {
                if (this == o)
                        return true;
                if (o == null)
                        return false;
                if (getClass() != o.getClass())
                        return false;
                return equalsTo((Hair) o);
        }

        // 1.c
        @Override
        public boolean equalsTo(Hair x) {
                return colors.equals(x.colors) && length == x.length;
        }
}

public enum Color {
        BROWN, DARK, BLONDE, RED, GRAY;
}
```

Si considerino ora i seguenti binding in Java:

```
Person alice = new Person("Alice", 23),
david = new Artist("Bowie", 69, new Hair(75, Set.of(Color.RED, Color.BROWN, Color.GRAY))),
Artist morgan = new Artist("Morgan", 47, new Hair(20, Set.of(Color.GRAY, Color.DARK))),
madonna = new Artist("Madonna", 60, new Hair(50, Set.of(Color.BLONDE)));
```

Per ciascuna delle seguenti espressioni Java si indichi con una crocetta l'esito della computazione - se produce un booleano, se non compila oppure se lancia un'eccezione a runtime:

a.	`alice.equals(null)`	◯ Si	◯ No
b.	`alice.equals(alice)`	◯ Si	◯ No
c.	`null.equals(david)`	◯ Si	◯ No
d.	`alice.equals(david)`	◯ Si	◯ No
e.	`alice.equalsTo(morgan)`	◯ Si	◯ No
f.	`morgan.equals(morgan)`	◯ Si	◯ No
g.	`morgan.equals(madonna)`	◯ Si	◯ No
h.	`morgan.equals(david)`	◯ Si	◯ No
i.	`morgan.equalsTo(david)`	◯ Si	◯ No
j.	`david.equalsTo(morgan)`	◯ Si	◯ No
k.	`madonna.equals(3)`	◯ Si	◯ No
l.	`madonna.equalsTo("Madonna")`	◯ Si	◯ No

1.3.2 Compare

1. Si assuma la firma del seguente metodo che ordina (stabile, crescente) una lista di elementi confrontabili:

```
static <T extends Comparable<T>> void sort(List<T> l)
```

(a) Se ipoteticamente definissimo un altro metodo `sort`, in overload con il precedente, avente firma:
`static <T> void sort(List<? extends Comparable<T>> l)`

esso rappresenterebbe un caso di overloading valido?

- ◯ Sì, perché i constraint del generic `T` sono diversi, quindi il compilatore sarebbe in grado di disambiguare.
- ◯ No, perché i metodi avrebbero la medesima *type erasure*.
- ◯ Sì, perché anche se l'overload è ambiguo per il compilatore, a runtime viene invocato il metodo giusto grazie al *dynamic dispatching*.
- ◯ No, perché qualunque overload che coinvolge generics sarebbe ambiguo a causa della *type erasure*.

Si considerino ora le seguenti classi:

```
public class Humanoid implements Comparable<Humanoid> {
        protected int strength;

        public Humanoid(int strength) { this.strength = strength; }

        @Override
        public int compareTo(Humanoid o) { return -(strength - o.strength); }
}

public class Elf extends Humanoid {
        protected int mana;

        public Elf(int strength, int mana) { super(strength); this.mana = mana; }

        @Override
        public int compareTo(Humanoid o) {
                if (o instanceof Elf) {
                        Elf e = (Elf) o;
                        return -((mana + strength) - (e.mana + e.strength));
                }
                else return super.compareTo(o);
        }
}
```

(b) Ipotizzando il codice seguente, la chiamata `sort(l)` sarebbe accettata dal compilatore Java?

```
List<Elf> l = new ArrayList<Elf>();
l.add(new Elf(11, 23));
sort(l);
```

- ◯ No: il constraint `<T extends Comparable<T>>` impone che `Elf` implementi `Comparable<Elf>`, ma invece implementa `Comparable<Humanoid>`.
- ◯ Sì: anche se il constraint `<T extends Comparable<T>>` impone che `Elf` implementi `Comparable<Elf>`, la chiamata risulta compatibile con `Comparable<Humanoid>` perché `Elf` è un sottotipo di `Humanoid`, da cui segue che `Comparable<Elf>` è sottotipo di `Comparable<Humanoid>`.
- ◯ No: sebbene il tipo dell'argomento sussuma in `List<Elf>`, l'interfaccia `List` non estende l'interfaccia `Comparable`.
- ◯ Sì: la chiamata sussume il tipo dell'argomento in `List<Humanoid>`, che soddisfa il constraint `<T extends Comparable<T>>` perché `Humanoid` implementa `Comparable<Humanoid>`.

(c) Si consideri un altro estratto di codice:

```
List<Humanoid> l = new ArrayList<>();
Humanoid a = new Elf(10, 8), b = new Humanoid(8), c = new Humanoid(12);
l.add(a); l.add(b); l.add(c);
sort(l);
```

Quale tra i seguenti identificatori è legato all'ultimo elemento della lista ordinata dopo la chiamata `sort(l)`?

◯ `a` ◯ `b` ◯ `c` ◯ non compila

2. Si prenda in considerazione il codice della domanda 1 (Sezione 1.3.1), il seguente statement sarebbe accettato dal compilatore Java 7+?

```
Artist c = Collections.max(artists, new Comparator<Person>() {
        public int compare(Person a, Person b) {
                return a.age - b.age;
        }
});
```

- ◯ No: il primo argomento `artists` ha tipo `List<Artist>`, istanziando il generic `T` con `Artist`, pertanto l'oggetto di tipo `Comparator<Person>` passato come secondo argomento non soddisfa il wildcard perché `Person` è diverso da `Artist`.
- ◯ Sì: il primo argomento `artists` ha tipo `List<Artist>`, istanziando il generic `T` con `Artist`, pertanto l'oggetto di tipo `Comparator<Person>` passato come secondo argomento soddisfa il wildcard con *lower bound* perché `Person` è supertipo di `Artist`.
- ◯ No: sebbene il primo argomento `artists` abbia tipo `List<Artist>`, il generic `T` viene istanziato col tipo `Person` in modo da soddisfare il wildcard con *upper bound*, tuttavia il tipo di ritorno `Person` non può essere *sussunto* in `Artist`.
- ◯ Sì: il primo argomento `artists` ha tipo `List<Artist>`, ma grazie al wildcard con *upper bound* il generic `T` viene istanziato col tipo `Person`; il compilatore tuttavia tiene traccia dei tipi degli oggetti a runtime, quindi il tipo di ritorno in realtà è `Artist`.

1.3.3 Programmazione ibrida funzionale

1. Si prendano in considerazione le seguenti interfacce Java:

```
public interface Solid extends Comparable<Solid> {
        double area();
        double volume();
        PositionedSolid at(Point origin);

        static <S extends Solid> int compareBy(Function<S, Double> f, S s1, S s2) {
                return Double.compare(f.apply(s1), f.apply(s2));
        }
        static <S extends Solid> Comparator<S> comparatorBy(Function<S, Double> f) {
                return (s1, s2) -> compareBy(f, s1, s2);
        }
        default int compareTo(Solid s) {
                return compareBy((x) -> x.volume(), this, s);
        }
}
```

La lambda espressione `(x) -> x.volume()` passata come primo argomento al metodo `compareBy()` nel corpo del metodo `compareTo()`, avente tipo `Function<Solid, Double>`, è equivalente a quale dei seguenti costrutti del linguaggio Java?

- ◯ Ad un riferimento ad un metodo statico (*static method reference*) della classe `Solid`; ovvero all'espressione `Solid::volume`.
- ◯ Ad un riferimento ad un metodo non-statico (*instance method reference*) di un oggetto arbitrario di tipo `Solid`; ovvero all'espressione `Solid::volume`.
- ◯ Ad un riferimento ad un metodo non-statico (*instance method reference*) di un oggetto specifico, che è `this` in questo caso; ovvero all'espressione `this::volume`.
- ◯ A nessuna dei precedenti.

2 Esercizi di progettazione e implementazione

I seguenti esercizi servono a verificare la capacità dello studente di progettare e implementare soluzioni utilizzando il paradigma a oggetti e il linguaggio Java.

1. Definiamo in Java 8+ un sistema di classi ed interfacce che rappresentano punti e solidi nello spazio $\mathbb{R}^3$. Per semplificare i calcoli legati alle coordinate assumiamo che tali solidi siano sempre orientati ortogonalmente rispetto agli assi cartesiani.
   (a) Si implementi un tipo che rappresenta punti tridimensionali *immutabili* nello spazio, ovvero una classe `Point` con i campi pubblici `x`, `y` e `z` di tipo `double` e l'opportuno costruttore.
   (b) Si implementi anche un metodo di `Point` avente firma `Point move(double dx, double dy, double dz)` che produce un nuovo punto *traslato* rispetto a `this` di dx, dy e dz rispettivamente per ogni dimensione.
   (c) Si prendano in considerazione le seguenti interfacce Java:

```
public interface Solid extends Comparable<Solid> {
    double area();
    double volume();
    PositionedSolid at(Point origin);

    static <S extends Solid> int compareBy(Function<S, Double> f, S s1, S s2) {
        return Double.compare(f.apply(s1), f.apply(s2));
    }
    static <S extends Solid> Comparator<S> comparatorBy(Function<S, Double> f) {
        return (s1, s2) -> compareBy(f, s1, s2);
    }
    default int compareTo(Solid s) {
        return compareBy((x) -> x.volume(), this, s);
    }
}

public interface Polyhedron extends Solid {
    double perimeter();
    @Override
    PositionedPolyhedron at(Point origin);
}

public interface PositionedSolid {
    Point origin();
}

public interface PositionedPolyhedron extends PositionedSolid, Iterable<Point> {}
```

   L'interfaccia `Solid` rappresenta oggetti tridimensionali *senza* una posizione specifica nello spazio; l'interfaccia `Polyhedron` rappresenta il sottoinsieme dei poliedri come sottotipi di `Solid` aventi un perimetro ed un numero finito di punti. Per posizionare un solido nello spazio è necessario invocare il metodo `at()`: il tipo `PositionedSolid` rappresenta solidi *con* una certa posizione nello spazio, detta origine. Il tipo `PositionedPolyhedron` specializza `PositionedSolid` aggiungendo ad un poliedro posizionato nello spazio la capacità di essere iterabile; l'iteratore deve fornire i punti di cui è costituito il poliedro, in un ordine qualunque.

   Si implementi la classe `Cube`, con il suo costruttore parametrico sulla lunghezza del lato ed i metodi richiesti dalle interfacce implementate.

```
public class Cube implements Polyhedron {
    private double side;    // lato del cubo
    /* implementare il resto */
}
```

   Suggerimento: si faccia particolare attenzione all'implementazione dell'iteratore di punti richiesto dall'interfaccia `Iterable<Point>` implementata dalla classe `PositionedPolyhedron`. Si usi il metodo `move()` di `Point` per calcolare *al volo* i vari punti di cui è costituito il cubo. Si consiglia infine di implementare il metodo `at()` tramite una anonymous class.

(d) Sarebbe possibile implementare il metodo `at()` tramite una lambda? ◯ Si ◯ No

(e) Si implementi la classe `Sphere`, con il suo costruttore parametrico sulla lunghezza del raggio ed i metodi richiesti dalle interfacce implementate[1]:

```
public class Sphere implements Solid {
    private double ray;     // raggio della sfera
    /* implementare il resto */
}
```

(f) Si prenda ora in considerazione questo *main* che costruisce alcune liste di solidi e poliedri:

```
public static void main(String[] args) {
    Cube cube1 = new Cube(11.), cube2 = new Cube(23.);
    Sphere sphere1 = new Sphere(12.), sphere2 = new Sphere(35.);
    List<Solid> solids = List.of(cube1, cube2, sphere1, sphere2);
    List<Cube> cubes = List.of(cube1, cube2);
    List<Sphere> spheres = List.of(sphere1, sphere2);
    List<? extends Polyhedron> polys = cubes;
}
```

i. Si ordini in ordine crescente la lista di cubi `cubes` secondo il valore del volume, scrivendo uno statement di invocazione del seguente metodo della classe `Collections` del JDK:
`static <T extends Comparable<? super T>> void sort(List<T> list)`

ii. Si ordini in ordine crescente la lista di sfere `spheres` secondo il valore della superficie totale, scrivendo uno statement di invocazione del seguente metodo della classe `Collections` del JDK:
`static <T> void sort(List<T> list, Comparator<? super T> c) {`
Bonus: si utilizzi il metodo `comparatorBy()` per ottenere l'oggetto `Comparator` desiderato.

iii. Per ciascuna dei seguenti binding Java si indichi con una crocetta l'esito della compilazione[2]:

`Comparator<Cube> cmpCube = Solid.comparatorBy(Cube::perimeter)`	◯ Si	◯ No
`Comparator<Solid> cmpSolid = Solid.comparatorBy(Solid::area)`	◯ Si	◯ No
`Comparator<Sphere> cmpSphere = Solid.comparatorBy(Sphere::perimeter)`	◯ Si	◯ No
`Comparator<Solid> cmpSolid2 = Solid.comparatorBy(Cube::area)`	◯ Si	◯ No
`Comparator<Polyhedron> cmpPoly = Solid.comparatorBy(Polyhedron::volume)`	◯ Si	◯ No
`Comparator<Sphere> cmpSphere2 = Solid.comparatorBy(Solid::area)`	◯ Si	◯ No
`Comparator<Polyhedron> cmpPoly2 = Solid.comparatorBy(Solid::volume)`	◯ Si	◯ No
`Comparator<Cube> cmpCube2 = Solid.comparatorBy(Polyhedron::perimeter)`	◯ Si	◯ No

iv. Per ciascuna delle seguenti espressioni Java si indichi con una crocetta l'esito della compilazione. Si assumano nello scope le variabili di tipo `Comparator` di cui all'esercizio precedente.

`Collections.sort(solids, cmpCube2)`	◯ Si	◯ No
`Collections.sort(cubes, cmpSolid)`	◯ Si	◯ No
`Collections.sort(spheres, cmpSphere2)`	◯ Si	◯ No
`Collections.sort(solids, cmpPoly2)`	◯ Si	◯ No
`Collections.sort(cubes, cmpSolid2)`	◯ Si	◯ No
`Collections.sort(cubes, cmpPoly)`	◯ Si	◯ No
`Collections.sort(spheres, cmpPoly2)`	◯ Si	◯ No
`Collections.sort(polys, cmpSolid)`	◯ Si	◯ No

v. Si implementi uno snippet di codice Java che, per ogni poliedro della lista `polys`, lo posiziona in un punto dello spazio a piacere e poi ne stampa in standard output tutti i punti.

2. Si implementi il seguente sistema di iteratori e trasformazioni tra iteratori.

(a) Si definisca una *interfaccia funzionale* simile a `java.util.BiFunction` che rappresenta funzioni binarie, parametrica sui 2 tipi degli argomenti e sul tipo di ritorno.

(b) Si definisca un tipo per la coppia eterogenea immutabile, ovvero una classe `Pair` parametrica su 2 tipi distinti che rappresentano rispettivamente il tipo del primo e del secondo elemento della coppia.

[1] Si rammenti che una sfera di raggio r ha volume $V = \frac{4}{3}\pi r^3$ e superficie totale $A = 4\pi r^2$.

[2] Ricordiamo la sintassi dei *method reference*: sia T un tipo e m il nome di un metodo non statico della classe T, allora l'espressione $T::m$ computa una funzione avente tanti parametri quanti sono i parametri nella firma del metodo m, più un parametro extra di tipo T che rappresenta l'oggetto sul quale invocare il metodo. Tale parametro extra di tipo T compare come primo parametro; il tipo di ritorno della funzione è lo stesso tipo di ritorno di m. Se m si riferisce ad un metodo non statico senza parametri, la funzione risultante è unaria ed ha solamente il parametro extra di tipo T. Ad esempio, si assuma un oggetto `c` di tipo `Cube`, allora il binding `Function<Cube, Double> f = Cube::area` è valido e l'espressione `f.apply(c)` ritorna l'area di `c` esattamente come l'invocazione diretta `c.area()`.

(c) Si definisca un tipo per la tripla eterogenea immutabile, ovvero una sottoclasse di `Pair` di nome `Triple`, parametrica su 3 tipi distinti che rappresentano i tipi dei 3 elementi.

(d) Si definisca un metodo statico `evalIterator` generico su 3 tipi `A`, `B` e `C` che, dato un iteratore su coppie $\texttt{A} * \texttt{B}$ ed una funzione binaria $\texttt{A} * \texttt{B} \to \texttt{C}$, produce un nuovo iteratore su triple $\texttt{A} * \texttt{B} * \texttt{C}$ che si comporta come un *wrapper* di quello in input. L'iteratore in output applica la funzione binaria ad ogni coppia di elementi letti dall'iteratore in input e produce una tripla con i due valori appena letti e passati alla funzione assieme al risultato di quest'ultima. Si utilizzino i tipi `Pair`, `Triple` e `BiFunction` definiti nei punti precedenti.

(e) Si implementi un classe `FiboSequence` le cui istanze rappresentano sequenze contigue di numeri di Fibonacci di lunghezza data in costruzione. Tali istanze devono essere *iterabili* tramite il costrutto *for-each* di Java, devono pertanto implementare l'interfaccia parametrica del JDK `java.util.Iterable<T>`. Ad esempio, il seguente codice deve compilare e stampare i primi 100 numeri di Fibonacci:

```
for (int n : new FiboSequence(100)) {
    System.out.println(n);
}
```

Requisito obbligatorio è che i numeri di Fibonacci generati dall'iteratore[3] sottostiano ad un meccanismo di *caching* che ne allevia il costo computazionale memorizzando il risultato di ogni passo di ricorsione, in modo che ogni computazione successiva con il medesimo input costi solamente un accesso in lettura alla cache.

(f) Si scriva il codice *de-zuccherato* dello statement *for-each* di cui al punto precedente.

(g) Si riscriva la classe `FiboSequence` in modo che la cache sia condivisa tra molteplici istanze.

(h) Si consideri una funzione `multiFact()` che data una `Collection<Integer>` produce una `Collection<FactThread>`, in modo che per ogni intero in input viene eseguito lo *spawn* di un nuovo thread che ne computa il fattoriale e ne conserva il risultato.

i. Si implementi la classe `FactThread` opportunamente, in modo che estenda `java.lang.Thread` ed incapsuli l'accesso al risultato della computazione del fattoriale.

ii. Si implementi la funzione `multiFact()`.

iii. Si scriva un pezzo di codice che testa la funzione `multiFact()` attendendo la terminazione di ciascun thread e stampando i risultati di ciascuno.

3. Si implementi in Java 8+ un sistema di classi che rappresentano punti, linee e poligoni regolarli nel piano cartesiano $\mathbb{R} \times \mathbb{R}$.

(a) Si implementi un tipo che rappresenta punti bidimensionali immutabili, ovvero una classe `Point` in cui le componenti `x` ed `y` sono di tipo `double`.

(b) Si implementi un tipo che rappresenta segmenti bidimensionali immutabili, ovvero una classe `Line` il cui costruttore prende due argomenti di tipo `Point`. Essa deve fornire un metodo `length()` che ne calcola la lunghezza tramite la distanza euclidea tra i due punti.

(c) Si prenda in considerazione la seguente classe astratta `Polygon` che rappresenta poligoni regolari come liste di punti (minimo 3, verificato a runtime). I punti nella lista determinano l'ordine di costruzione dei segmenti di cui è composto il poligono. Ad esempio, una lista contenente i seguenti 3 punti $A = (0, 0)$, $B = (3, 3)$ e $C = (3, 0)$ rappresenta un triangolo rettangolo in cui il primo lato è AB, il secondo è BC ed il terzo è CA.

```
public abstract class Polygon {
    protected final List<Point> points;
    protected Polygon(List<Point> points) {
        assert points.size() >= 3;
        this.points = points;
    }
    public Iterator<Line> lineIterator() { /* da implementare */ }
    public double perimeter() { /* da implementare */ }
    public abstract double area();
}
```

i. Per quale motivo è necessario vincolare a runtime la dimensione minima della lista di punti tramite un `assert` anziché sfruttare in qualche modo il type system per fare un controllo statico? Si articoli una breve risposta.

[3] Nell'esempio l'iteratore è implicitamente utilizzato dal costrutto *for-each*.

ii. Si implementi il metodo `lineIterator()` che costruisce un iteratore su oggetti di tipo `Line` e si comporta come un *wrapper* dell'iteratore estratto dal campo `points`. Gli oggetti prodotti dall'iteratore di `Line` devono essere costruiti *dinamicamente* leggendo coppie di punti adiacenti dall'iteratore di `Point`. Si implementi una logica di *caching* dell'ultimo punto letto per permettere la costruzione di un nuovo segmento adiacente all'ultimo ad ogni invocazione del metodo `next()`. Si badi inoltre a riusare opportunamente il primo punto come secondo estremo dell'ultimo segmento costruito.

iii. Si implementi il metodo `perimeter()` in funzione del metodo `lineIterator()`, ovvero calcolando il perimetro del poligono iterando sui segmenti che lo compongono.

(d) Estendiamo ora la gerarchia di classi introducendo tipi specializzati per i poligoni classici.

i. Si implementi una sottoclasse di `Polygon` di nome `Triangle` che rappresenta triangoli qualunque.

```
public class Triangle extends Polygon {
    public Triangle(Point p1, Point p2, Point p3) { /* da implementare */ }
    @Override
    public double area() { /* da implementare */ }
}
```

Il costruttore prende 3 argomenti di tipo `Point` e deve chiamare il super-costruttore opportunamente. Si implementi il metodo `area()` in modo che calcoli l'area del triangolo senza fare assunzioni sulla sua forma.

ii. Si implementi un tipo che rappresenta triangoli rettangoli tramite una sottoclasse di `Triangle`. Si badi in particolar modo a definire un costruttore che sintetizzi al massimo le informazioni passate come argomenti. Si prediligano i controlli statici a quelli dinamici, se possibile, e si tenti di minimizzare i casi di ambiguità o gli stati di invalidità dell'oggetto; nel caso in cui si ritenga necessario fare dei controlli a runtime, si usi il costrutto `assert`.

iii. Si implementi una sottoclasse di `Polygon` di nome `Rectangle` che rappresenta rettangoli.

```
public class Rectangle extends Polygon {
    public Rectangle(Point p1, Point p3) { /* da implementare */ }
    @Override
    public double area() { /* da implementare */ }
}
```

Il costruttore prende i 2 punti della diagonale e deve passare al super-costruttore i 4 punti che costituiscono il rettangolo calcolandone le coordinate per proiezione ortogonale. Si badi all'ordine in cui i punti compaiono nella lista, affinché siano adiacenti e permettano di comporre i lati correttamente. Si implementi il metodo `area()` in modo che calcoli l'area del rettangolo.

iv. Si implementi una sottoclasse di `Rectangle` di nome `Square` che rappresenta quadrati.

```
public static class Square extends Rectangle {
    public Square(Point p1, double side) { /* da implementare */ }
}
```

Il costruttore prende il punto in basso a sinistra e la dimensione del lato e deve chiamare il super-costruttore opportunamente.

(e) Si prenda in considerazione il seguente codice, in cui compare l'invocazione di un metodo statico `max` da definire:

```
Square sq1 = new Square(new Point(10., -4.), 0.1),
       sq2 = new Square(new Point(1., 20.), 0.01);
Collection<Square> squares = List.of(sq1, sq2);
Rectangle r = max(squares, new Comparator<Polygon>() {
    @Override
    public int compare(Polygon a, Polygon b) {
        return (int) (a.area() - b.area());
    }
});
```

i. Si scriva la firma e l'implementazione del metodo statico `max()` invocato nell'ultimo statement affinché il codice di cui sopra compili correttamente. Si renda tale metodo più generico possibile e non monomorfo rispetto ai tipi che compaiono in questa invocazione, prestando particolare attenzione ai vincoli sui generics. La semantica di `max` è facilmente intuibile: trova l'elemento maggiore usando il comparatore per confrontare gli elementi della collection di input.

ii. Assumendo il comportamento corretto del metodo `max()`, quale delle seguenti espressioni booleane è vera?

◯ `sq1 == r` ◯ `sq2 == r` ◯ `r == null` ◯ nessuna delle precedenti

iii. Quale dei seguenti numeri razionali rappresenta il valore di tipo `double` computato dall'espressione `r.area()`?
◯ 10^{-1} ◯ 10^{-2} ◯ 10^{-4} ◯ non è un quadrato ma un rettangolo

4. Si consideri la seguente interfaccia parametrica in linguaggio Java:

```
public interface Equatable<T> {
    boolean equalsTo(T x);
}
```

Essa non sostituisce il meccanismo basato sul metodo `equals(Object)` della classe `Object`, tuttavia permette di implementare il confronto di uguaglianza in maniera più sicura delegando parte della logica ad un metodo fortemente tipato.

Si consideri ora la classe parametrica `Person`, che supporta il confronto di uguaglianza con oggetti di tipo `P`:

```
public class Person<P extends Person<P>> implements Equatable<P> {
    public final String name;
    public final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public boolean equals(Object o) { /* da implementare */ }

    @Override
    public boolean equalsTo(P other) { /* da implementare */ }

    @Override
    public String toString() { return name; }
}
```

(a) Si implementi il metodo `equals(Object)` della classe `Person` in modo che, dati `p1` e `p2` di tipo `Person`, le seguenti invarianti siano rispettate:

- `p1.equals(p1) == true`
- `p1.equals(null) == false`
- `p1.equals(e) == false` se l'espressione `e` ha tipo diverso dal tipo di `p1`[4]
- `p1.equals(p2) == p1.equalsTo(p2)`

(b) Si implementi il metodo `equalsTo(P)` della classe `Person` in modo che due oggetti siano uguali quando hanno il medesimo nome e la medesima età.

Si consideri ora il seguente codice:

```
public class Artist extends Person<Artist> {
    public final Hair hair;

    public Artist(String name, int age, Hair hair) {
        super(name, age);
        this.hair = hair;
    }

    @Override
    public boolean equalsTo(Artist other) { /* da implementare */ }
}
```

[4] Ricordiamo che il metodo della classe `Object` avente firma `Class<? extends Object> getClass()` consente di estrarre a runtime il tipo *raw* di un oggetto.

```
public class Hair implements Equatable<Hair> {
    public final int length;
    public final Set<Color> colors;

    public Hair(int length, Set<Color> colors) {
        this.colors = colors;
        this.length = length;
    }

    @Override
    public boolean equals(Object o) { /* da implementare */ }

    @Override
    public boolean equalsTo(Hair x) { /* da implementare */ }
}

public enum Color {
    BROWN, DARK, BLONDE, RED, GRAY;
}
```

(c) Si implementino i metodi `equals(Object)` e `equalsTo(Hair)` della classe `Hair` in modo che due oggetti siano uguali quando hanno la medesima lunghezza ed i medesimi colori (indipendentemente dal loro ordine). Si rispettino le stesse invarianti del punto (a) per l'implementazione del metodo `equals(Object)` e si deleghi, come per la classe `Person`, la parte fortemente tipata del confronto al metodo `equalsTo(Hair)`.

(d) Il metodo `equalsTo(Artist)` è davvero un *override* del metodo `equalsTo(P)` ereditato dalla superclasse?

- ◯ No, perché la *type erasure* elimina il generic `P` che viene sostuito col suo constraint `Person` nella superclasse, la quale introduce di fatto un metodo `equalsTo(Person)`, di cui `equalsTo(Artist)` non è un override ma un overload.
- ◯ Sì, perché il metodo `equalsTo()` viene gestito in modo particolare dal compilatore Java, permettendo override anche con firme differenti.
- ◯ No, perché la firma è diversa e quindi è un overload, non un override.
- ◯ Sì, perché il generic `P` viene sostituito col tipo `Artist` nello scope della sottoclasse, pertanto viene ereditato un metodo `equalsTo(Artist)`, rendendo questo un vero override.

(e) Si implementi il metodo `equalsTo(Artist)` della classe `Artist` in modo che due oggetti siano uguali quando hanno i medesimi nome, età e capelli. Si badi a riusare opportunamente l'implementazione ereditata dalla superclasse.

Si considerino ora i seguenti binding in Java:

```
Person alice = new Person("Alice", 23),
       david = new Artist("Bowie", 69, new Hair(75, Set.of(Color.RED, Color.BROWN, Color.GRAY)));
Artist morgan = new Artist("Morgan", 47, new Hair(20, Set.of(Color.GRAY, Color.DARK))),
       madonna = new Artist("Madonna", 60, new Hair(50, Set.of(Color.BLONDE)));
```

(g) Si prenda in considerazione il seguente metodo della classe `java.util.Collections` del JDK 7+ per computare l'elemento massimo in una collection secondo un criterio di ordinamento dato:

```
static <T> T max(Collection<? extends T> c, Comparator<? super T> cmp)
```

Si considerino ora i seguenti binding in Java:

```
List<Artist> artists = Arrays.asList((Artist) david, morgan, madonna);
List<Person> persons = Arrays.asList(alice, david, morgan, madonna);
```

i. Si scriva uno statement di invocazione del metodo `max()` che computi l'oggetto della lista `artists` avente il più alto prodotto tra lunghezza dei capelli e numero di colori.

ii. Si scriva uno statement di invocazione del metodo `max()` che computi l'oggetto della lista `persons` avente il nome che viene per primo in ordine lessicografico.

iii. Il seguente statement sarebbe accettato dal compilatore Java 7+?

```
Artist c = Collections.max(artists, new Comparator<Person>() {
            public int compare(Person a, Person b) {
                return a.age - b.age;
            }
        });
```

(h) Si implementi una classe `Node`, parametrica su un tipo generico `T`, che rappresenta nodi di un albero binario decorati con valori di tipo `T`. I puntatori ai due sottoalberi sinistro e destro possono essere `null`, per rappresentare l'assenza di un sottoalbero o di entrambi; un nodo con entrambi i sottoalberi nulli rappresenta una foglia. La classe `Node` deve essere *iterabile* e l'iteratore deve attraversare l'albero in DFS (*Depth-First Search*), fornendo gli elementi nell'ordine in cui li incontra discendendo nei sottorami, prediligendo la ricorsione in profondità prima nel sottoalbero sinistro e poi in quello destro.
Suggerimento: si utilizzi una collection di appoggio da popolare durante la visita dell'albero.

5. Si implementi il seguente sistema di iteratori e trasformazioni tra iteratori.

   (a) i. Si definisca una *interfaccia funzionale* simile a quella definita in `java.util.Function` parametrica sia sul tipo dell'argomento che sul tipo di ritorno.

   ii. Si definisca un metodo statico `mapIterator` generico su due tipi `A` e `B` che, dato un iteratore su `A` ed una funzione da `A` a `B`, produce un nuovo iteratore che si comporta come un *wrapper* di quello in input applicando la funzione ad ogni elemento iterato. Si utilizzi il tipo funzione definito nel punto precedente.

   (b) i. Si definisca un tipo per la coppia eterogenea, ovvero una classe `Pair` parametrica su due tipi distinti `A` e `B` che rappresentano rispettivamente il tipo del primo e del secondo elemento della coppia.

   ii. Si definisca un metodo statico `pairIterator` generico su un tipo `A` che, dato un iteratore su `A`, ritorna un nuovo iteratore che itera su coppie di elementi, prendendone due a due dall'iteratore passato come argomento. Quando non sono disponibili 2 elementi, si consideri la sequenza come terminata - in altre parole, una sequenza di lunghezza dispari viene trattata come una sequenza di coppie fino al penultimo elemento, scartando l'ultimo. Si utilizzi il tipo `Pair` definito nel punto precedente.

   (c) Si scriva un blocco di codice Java che utilizza i metodi di cui sopra per trasformare un iteratore di cateti in un iteratore di ipotenuse che applica il teorema di pitagora in tempo reale. In particolare, si scriva un blocco di codice che:

   1. crea una collection iterabile a vostra scelta e la popola con numeri interi casuali usando la classe `Random` del JDK[5];
   2. produce un iteratore su coppie di interi, avente tipo `Iterator<Pair<Integer, Integer>>`, passando l'iteratore originale della collection come argomento al metodo `pairIterator` definito nell'esercizio precedente;
   3. trasforma l'iteratore su coppie di interi appena prodotto in un altro iteratore che itera su `Double` chiamando `mapIterator` con una callback opportunamente definita. Tale callback calcola l'ipotenusa di un triangolo rettangolo data una coppia di cateti. Si badi che i cateti sono interi e le ipotenuse sono `double`.

6. Si prenda in considerazione la seguente interfaccia Java:

```
public interface Pool<T, R> {
    void add(T x);                              // popola la pool con un nuovo elemento
    R acquire() throws InterruptedException;    // acquisisce una risorsa
    void release(R x);                          // rilascia una risorsa e la rimette nella pool
}
```

Una pool è un container di oggetti che si comporta come una coda bloccante: è possibile ottenere una risorsa con `acquire()` per poi restituirla alla pool tramite il metodo `release()`.

Il generic `T` astrae il tipo degli oggetti contenuti internamente nella pool, mentre `R` è il tipo della risorsa restituita dalla `acquire()`: il motivo per cui sono due generic differenti è per consentire alle classi che implementano questa interfaccia di rappresentare in modo diverso, se necessario, gli elementi conservati all'interno e le risorse restituite dalla `acquire()`.

Quando la coda è vuota e nessun oggetto è disponibile, il metodo `acquire()` deve essere bloccante: al fine di semplificare l'implementazione, si utilizzi un oggetto di tipo `LinkedBlockingQueue` come campo interno. Riportiamo un estratto della classe `LinkedBlockingQueue` definita dal JDK con i metodi pubblici più significativi:

[5] Ricordiamo che la classe `java.util.Random` offre un costruttore senza parametri per inizializzare il PRNG e dei metodi `nextInt()` e `nextDouble()` per generare, rispettivamente, un intero o un double compresi tra 0 e il valore massimo.

```
class LinkedBlockingQueue<E> implements BlockingQueue<E> {
    LinkedBlockingQueue();                  // costruttore
    void add(E x);                          // aggiunge un elemento
    E take() throws InterruptedException;   // estrae la testa (bloccante)
    E peek();                               // ritorna la testa senza rimuoverla,
                                            // oppure null se vuota

    int size();                             // numero di elementi
    // etc...
}
```

(a) Si definisca una interfaccia `BasicPool` che estende l'interfaccia `Pool` e che ha un solo generic per rappresentare sia il tipo degli elementi interni sia il tipo delle risorse restituite dalla `acquire()`.

(b) Si implementi una classe `SimplePool` che implementa l'interfaccia `BasicPool` e realizza una semplice coda bloccante.

(c) Si implementi una classe `AutoPool` che implementa l'interfaccia `Pool` realizzando un meccanismo di *auto-release* delle risorse. Se `p` è una pool, l'obiettivo è fare in modo che una risorsa `x` acquisita tramite la chiamata `p.acquire()` non debba essere esplicitamente riconsegnata alla pool chiamando `p.release(x)`, ma sia possibile rilasciarla invocando `x.release()`.

Suggerimento: si definisca un nuovo tipo parametrico `Resource` che si comporta come un *proxy* per l'oggetto contenuto al suo interno e che implementa la logica di *auto-release* rilasciando `this`.

(d) Si automatizzi il meccanismo di cui sopra facendo in modo che un oggetto di tipo `Resource` si rilasci *autonomamente* quando non esistono più riferimenti ad esso.

Suggerimento: la superclasse `Object` definisce un metodo `finalize()` che viene invocato nel momento in cui l'oggetto viene cancellato dal garbage collector.

7. Si prenda in considerazione questa semplificazione della classe `java.util.Random` del JDK: essenzialmente essa offre un costruttore senza parametri, un secondo costruttore con il seed per inizializzare il PRNG ed alcuni metodi per generare valori numerici di tipo differente:

```
public class Random {
    public Random() { ... }
    public Random(int seed) { ... }
    public boolean nextBoolean() { ... }
    public int nextInt() { ... }
    public double nextDouble() { ... }
}
```

(a) Si scriva un *wrapper* della classe `Random` che si comporta come un **singleton**, facendo attenzione a riprodurre ogni aspetto dell'originale.

(b) Si definisca una classe `RandomIterator` che implementa l'interfaccia `java.util.Iterator<Integer>` del JDK e che si comporta come un iteratore su interi, generando un numero casuale ad ogni invocazione del metodo `next()` anziché scorrendo una vera collection, fino ad esaurire la sequenza di lunghezza specificata in costruzione. Si implementino opportunamente il costruttore ed i metodi richiesti dall'interfaccia.

8. Si scriva un metodo statico e generico `compareMany` che, dati due parametri di tipo `java.util.Collection` generici su due tipi differenti, confronti ogni elemento di tipo `A` della prima con il corrispettivo elemento di tipo `B` della seconda. Il confronto tra elementi va implementato chiamando il metodo `compareTo` opportunamente; qualsiasi elemento differente rende le collection differenti, così come una lunghezza diversa. Il risultato del metodo `compareMany` è di tipo `int` e deve rispettare la semantica del confronto *a tre vie* di Java, da reinterpretare in modo ragionevole per il caso specifico del confronto tra container.

9. Si implementi una sottoclasse generica di `java.util.ArrayList` di nome `SkippableArrayList` che estende la superclasse con un iteratore in grado di discriminare gli elementi secondo un predicato booleano. Gli elementi che soddisfano il predicato vengono processati da una certa funzione di trasformazione[6]; gli altri vengono passati ad una seconda callback (non una funzione di trasformazione).

[6] Una funzione di trasformazione è una funzione in cui dominio è uguale al codominio, per esempio una funzione $f : \tau \to \tau$ è una funzione di trasformazione sull'insieme τ.

(a) Si definisca una *interfaccia funzionale* di nome `Predicate` specializzando l'interfaccia generica `java.util.Function` del JDK in modo che il tipo del parametro dell metodo `apply` sia generico ed il tipo di ritorno sia `Boolean`.

(b) Si definisca una interfaccia di nome `Either` parametrica su un tipo `T` che include due metodi di nome diverso: il primo metodo, `onSuccess`, è una funzione di trasformazione che viene chiamata dall'iteratore quando il predicato ha successo; il secondo metodo, `onFailure`, viene invocato invece quando il predicato fallisce, prende un argomento di tipo `T` e non produce alcun risultato, tuttavia può lanciare una eccezione di tipo `Exception`.

(c) Si definisca la sottoclasse `SkippableArrayList` parametrica su un tipo `E` e si implementi un metodo pubblico avente firma `Iterator<E> iterator(Predicate<E> p, Either<E> f)` che crea un iteratore con le caratteristiche accennate sopra. In particolare:

- l'iteratore parte sempre dall'inizio della collezione ed arriva alla fine, andando avanti di un elemento alla volta normalmente;
- ad ogni passo l'iteratore applica il predicato `p` all'elemento corrente: se il predicato `p` viene soddisfatto allora viene invocato il metodo `onSuccess` di `f` e passato l'elemento corrente come argomento; altrimenti viene invocato il metodo `onFailure` e passato l'elemento corrente come argomento a quest'ultimo;
- l'invocazione di `onFailure` deve essere racchiusa dentro un blocco che assicura il *trapping* delle eccezioni - in altre parole, una eccezione proveniente dall'invocazione di `onFailure` non deve interrompere lo scorrimento della collection da parte dell'iteratore;
- quando viene invocato `onSuccess`, il suo risultato viene restituito come elemento corrente dall'iteratore;
- quando viene invocato `onFailure`, l'iteratore ritorna l'elemento originale che ha fatto fallire il predicato.

(d) Si scriva un esempio di codice `main` che:

- costruisce una `ArrayList` di interi vuota;
- costruisce una `SkippableArrayList` di interi;
- popola quest'ultima con numeri casuali compresi tra 0 e 10, inclusi gli estremi[7];
- invocando **solamente una volta** il metodo `iterator(Predicate<E>, Either<E>)` della `SkippableArrayList` con gli argomenti opportuni, somma 1 a tutti gli elementi maggiori di 5 e appende all'`ArrayList` quelli minori o uguali a 5.

10. Si implementi in Java una sottoclasse generica di `ArrayList` di nome `FancyArrayList` che estende le funzionalità della superclasse con un iteratore più versatile. Il nuovo iteratore deve essere in grado di andare sia avanti che indietro secondo un valore di incremento intero non nullo; e di processare gli elementi che incontra durante l'attraversamento applicando una funzione di trasformazione[8].

(a) Si definisca una *interfaccia funzionale* di nome `Function` parametrica sia sul tipo del dominio che sul tipo del codominio, equivalente a quella definita dal JDK 8+ nel package `java.util.function`.

(b) Si definisca la sottoclasse `FancyArrayList` parametrica su un tipo `E` e si implementi un metodo pubblico avente firma `Iterator<E> iterator(int step, Function<E, E> f)` che crea un iteratore con le caratteristiche accennate sopra tramite una *classe anonima*. Più precisamente:

- quando il valore del parametro `step` è positivo, l'iteratore parte dall'inizio della collezione e va *avanti* incrementando il cursore di `step` posizioni ad ogni passo; quando invece `step` è negativo, l'iteratore parte dalla fine della collezione e va *indietro* decrementando il cursore;
- ad ogni passo l'iteratore applica la funzione di trasformazione `f` all'elemento da restituire.

(c) Si aggiungano a `FancyArrayList` i seguenti metodi pubblici, badando ad implementarli in funzione del metodo `iterator(int, Function<E, E>)` realizzato per l'esercizio precedente, *senza replicazioni* di codice. Per ciascuno si specifichi inoltre se è un override oppure no, utilizzando opportunamente l'annotazione `@Override`.

i. Si implementi il metodo avente firma `Iterator<E> iterator()` che produce un iteratore convenzionale che procede in avanti di una posizione alla volta.

ii. Si implementi il metodo avente firma `Iterator<E> backwardIterator()` che produce un iteratore rovescio che procede indietro di una posizione alla volta.

(d) Si rifattorizzi il metodo `iterator(int, Function<E, E>)` realizzato per il punto (b) in modo che non usi una classe anonima, ma una nuova classe innestata *statica* e parametrica di nome `FancyIterator`. Si presti particolare attenzione all'uso dei generics ed al passaggio esplicito della *enclosing instance* al costruttore.

[7] Si utilizzi la classe `Random` del JDK: il costruttore non ha parametri ed il metodo per generare un intero tra 0 ed `n` (esclusivo) ha firma `nextInt(int n)`.

[8] Una funzione di trasformazione è una funzione in cui dominio è uguale al codominio, per esempio $f : \tau \to \tau$ per un qualche insieme τ.

11. Si realizzi in Java un algoritmo che trova il minimo ed il massimo in una lista generica e restituisca i risultati, rispettivamente, come primo e secondo elemento di una coppia omogenea.

    (a) Si definisca un tipo per la coppia eterogenea, ovvero una classe `Pair` parametrica su due tipi distinti `A` e `B` che rappresentano rispettivamente il tipo del primo e del secondo elemento della coppia.

    (b) Si implementi un metodo statico e generico avente la seguente firma[9]:

    ```
    static <E> Pair<E, E> findMinAndMax(List<E> l, Comparator<E> c)
    ```

    L'algoritmo di ricerca del minimo e del massimo deve eseguire *una sola traversata* della lista; si assuma che gli argomenti siano non-nulli e che la lista abbia sempre almeno un elemento.

    (c) Si definisca un metodo in overload con il precedente che non usi un `Comparator` ma aggiunga il constraint `Comparable` al generic `E`, secondo la seguente firma:

    ```
    static <E extends Comparable<E>> Pair<E, E> findMinAndMax(List<E> l)
    ```

    Si implementi questo metodo in funzione del precedente, *senza replicare* l'algoritmo di ricerca.

12. Si assumano le seguenti classi ed interfacce Java, innestate per brevità in una sola classe a top-level:

```java
public class HannaBarbera {

    public interface Food {
        int getWeight();
    }

    public interface Animal extends Food {
        void eat(Food f);
    }

    public static class Mouse implements Animal {
        private int weight;

        public Mouse(int weight) { this.weight = weight; }

        @Override
        public void eat(Food f) { weight += f.getWeight() / 2; }

        @Override
        public int getWeight() { return weight; }
    }

    public static class Cat implements Animal {
        private int weight;

        public Cat(int weight) { this.weight = weight; }

        @Override
        public void eat(Food f) { weight += f.getWeight() / 5; }

        @Override
        public int getWeight() { return weight; }
    }

    public static class Cheese implements Food {
        @Override
        public int getWeight() { return 50; }
    }
}
```

[9] L'interfaccia parametrica `Comparator<T>` include un metodo binario avente firma `int compare(T a, T b)`, il quale confronta i due argomenti e ritorna un intero secondo la semantica standard del confronto *a tre vie* in Java.

(a) Come sarebbe possibile rifattorizzare questo codice in modo da ridurre al minimo le ripetizioni e massimizzare il riuso? Si scriva il codice rifattorizzato, facendo attenzione a non cambiare il comportamento del programma.

(b) Si prenda in considerazione il seguente metodo `main()` per la classe `HannaBarbera`, supponendo che le restanti classi ed interfacce innestate siano quelle rifattorizzate secondo quanto richiesto dall'esercizio 1.

```
public static void main(String[] args) {
    Animal tom = new Cat(200);
    Animal jerry = new Mouse(10);
    jerry.eat(new Cheese());
    jerry.eat(new Food() {
        @Override
        public int getWeight() { return 10; }
    });
    tom.eat(jerry);
    System.out.println(String.format("Tom now weights %d", tom.getWeight()));
}
```

i. Qualora una o più parti del metodo `main()` non siano compatibili a causa della rifattorizzazione eseguita precedentemente, si scriva in maniera chiara quali modifiche sono necessarie per renderlo compilabile mantenendo la semantica dell'originale; se preferibile, si riscriva per chiarezza l'intero codice `main()` opportunamente modificato.

ii. Qual'è il peso del gatto Tom che viene scritto in output alla fine?

13. Si consideri il seguente codice Java, che definisce:

- un tipo `Point` che rappresenta un punto bidimensionale immutabile;
- un supertipo `Line` che rappresenta un oggetto geometrico unidimensionale astratto in grado di supportare il confronto con oggetti del medesimo tipo;
- un sottotipo `Segment` che rappresenta un segmento;
- un sottotipo `Polyline` che rappresenta una spezzata.

```
public class Geometry {

    public static class Point {
        public final double x, y;
        public Point(double x, double y) {
            this.x = x;
            this.y = y;
        }
    }

    public static abstract class Line<S extends Line<S>> implements Comparable<S> {
        public abstract double length();

        @Override
        public int compareTo(S l) { return Double.compare(length(), l.length()); }
    }

    public static class Segment extends Line<Segment> {
        private Point p1, p2;

        public Segment(Point p1, Point p2) {
            this.p1 = p1;
            this.p2 = p2;
        }

        @Override
        public double length() {
            // da implementare
```

```
        }
    }

    public static class Polyline extends Line<Polyline> {
        private List<? extends Point> points;

        public Polyline(List<? extends Point> points) { this.points = points; }

        @Override
        public double length() {
            // da implementare
        }
    }
}
```

(a) Si implementi il metodo `length()` in override nella classe `Segment` in modo che calcoli la lunghezza del segmento passante per i due punti memorizzati nei campi privati dell'oggetto.

(b) Si implementi il metodo `length()` in override nella classe `Polyline` in modo che calcoli la lunghezza della spezzata come sommatoria delle lunghezze dei segmenti di cui è costituita; tali segmenti sono ricostruibili prendendo le coppie di punti consecutive contenute nella lista di punti memorizzata nell'oggetto.

(c) Si consideri il seguente `main()` per la classe `Geometry`: vengono create due spezzate di tipo `Polyline` a partire da dei punti tridimensionali; le spezzate vengono poi confrontate tramite una funzione `max()` e viene scritta in output la lunghezza della maggiore[10].

```
public static void main(String[] args) {
    Point a = new Point3D(0., 0., 0.),
          b = new Point3D(1., 1., 1.),
          c = new Point3D(2., 0., 3.);

    Polyline abc = new Polyline(List.of(a, b, c)),
             bac = new Polyline(List.of(b, a, c));

    System.out.println(String.format("max length = %g", max(abc, bac).length()));
}
```

i. Definire la classe `Point3D` che estende `Point` ed aggiunge la terza dimensione `z` al punto. Si mantenga lo stesso stile, detto *immutabile*, senza getter né setter.

ii. Si implementi il metodo statico `max()`, binario e generico su un tipo `T`, che calcola il massimo tra due oggetti che rispettano l'interfaccia `Comparable<T>` del JDK.

iii. Si indichi qual'è, tra i seguenti numeri irrazionali, quello che rappresenta il valore esatto che l'espressione `max(abc, bac).length()` computa approssimando ad un numero in virgola mobile di tipo `double`.
◯ $2\sqrt{2}$ ◯ $2\sqrt{2}+1$ ◯ $\sqrt{2}+2$ ◯ $2\sqrt{2}+2$

[10] Il metodo `List.of()` è definito nella classe parametrica `java.util.List` del JDK e produce una lista popolata con gli oggetti passati come argomenti. La firma è: `static <E> List<E> of(E e1, E e2, E e3)`.

3 Domande aperte

Le seguenti domande servono a verificare le conoscenze dello studente relativamente agli argomenti coperti durante il corso. Lo studente deve dimostrare di aver studiato l'argomento ed essere in grado di esporlo in modo adeguato.

1. *Incapsulamento*: spiegare cosa si intende per metodo setter. Fornire almeno una motivazione per la quale l'uso di setter è preferibile alla definizione di campi pubblici. Fare un esempio concreto (con codice) di un setter che svolge una funzione non ottenibile con un campo pubblico.
2. *Ereditarietà*: spiegare approfonditamente in concetto di sovrascrittura di un metodo e quali sono le condizioni nelle quali può essere utile. Fare un esempio pratico mostrando le necessarie porzioni di codice.
3. *Espressioni Lambda*: considerare l'interfaccia `ActionListener` con l'unico metodo `void actionPerformed(ActionEvent e)`. Considerare il `JButton b` e mostrare (con codice) come utilizzare il metodo `void addActionListener(ActionListener l)` per aggiungere un `ActionListener` che stampi a console la stringa `"Java"`, codificato rispettivamente come una classe anonima e come un'espressione Lambda.

4 Soluzioni

4.1 Soluzione Tests

Sezione 1.1:

1. Singleton
2. Perché i costruttori non sono metodi soggetti al *dynamic dispatching*, quindi non è possibile sfruttare il polimorfismo per costruire oggetti se non tramite un metodo non-statico che incapsula la costruzione.

Sezione 1.2:

1. No, però non c'è limite al numero di interfacce che una classe può implementare.
2. A riusare lo stesso codice, ovvero chiamare un metodo, con parametri di tipo differente.
3. $\varphi \preceq \rho$
4. No, ma rende possibile il polimorfismo subtype poiché garantisce che i membri pubblici delle superclassi esistano nelle sottoclassi.
5. Il tipo ed il numero dei parametri e basta, senza tipo di ritorno né eccezioni.
6. Sì, anche se le sottoclassi `Paladin` e `Rogue` specializzano il tipo di ritorno rispetto al type argument passato alla superclasse, la specializzazione del tipo di ritorno è supportata dall'overriding in Java.
7. `static <A, B> Collection<B> map(Collection<A> c, Function<? super A, ? extends B> f)`
8. (a) `Paladin` il compilatore è in grado di discriminare.
   (b) Sì, compilerebbe e risolverebbe `normalizeAttack(Paladin)`.
9. L'overriding supporta la *co-varianza* del tipo di ritorno di un metodo.
10. No: l'overloading non è possibile tra metodi che differiscono solamente per il tipo di ritorno;

Sezione 1.3.1:

1. (a) false
   (b) true
   (c) non compila/eccezione[11]
   (d) false
   (e) false
   (f) true

[11] alcuni compilatori rifiutano `null.metodo()`, altri lo accettano ma a runtime viene lanciato `NullPointerException`

(g) false

(h) false

(i) non compila

(j) false

(k) false

(l) non compila

Sezione 1.3.2:

1. (a) No, perché i metodi avrebbero la medesima *type erasure*.

   (b) No: il constraint `<T extends Comparable<T>>` impone che `Elf` implementi `Comparable<Elf>`, ma invece implementa `Comparable<Humanoid>`.

   (c) `b`

2. Sì: il primo argomento `artists` ha tipo `List<Artist>`, istanziando il generic `T` con `Artist`, pertanto l'oggetto di tipo `Comparator<Person>` passato come secondo argomento soddisfa il wildcard con *lower bound* perché `Person` è supertipo di `Artist`.

Sezione 1.3.3:

1. Ad un riferimento ad un metodo non-statico (*instance method reference*) di un oggetto specifico, che è `this` in questo caso; ovvero all'espressione `this::volume`.

4.2 Soluzione esercizi di progettazione e implementazione

4.2.1 Esercizio 1

```java
import java.util.*;
import java.util.function.Function;

public class Es1 {

    // 1.a
    public static class Point {
        public final double x, y, z;

        public Point(double x, double y, double z) {
            this.x = x;
            this.y = y;
            this.z = z;
        }

        public Point move(double dx, double dy, double dz) {
            return new Point(x + dx, y + dy, z + dz);
        }

    }

    // 1.b
    public interface Solid extends Comparable<Solid> {
        double area();
        double volume();
        PositionedSolid at(Point origin);

        static <S extends Solid> int compareBy(Function<S, Double> f, S s1, S s2) {
            return Double.compare(f.apply(s1), f.apply(s2));
        }
```

```
    static <S extends Solid> Comparator<S> comparatorBy(Function<S, Double> f) {
        return (s1, s2) -> compareBy(f, s1, s2);
    }

    default int compareTo(Solid s) {
        return compareBy((x) -> x.volume(),this, s);
    }
}

public interface PositionedSolid {
    Point origin();
}

public interface Polyhedron extends Solid {
    double perimeter();

    @Override
    PositionedPolyhedron at(Point origin);
}

public interface PositionedPolyhedron extends PositionedSolid, Iterable<Point> {
}

// 1.c
public static class Cube implements Polyhedron {
    private double side;    // lato del cubo

    public Cube(double side) {
        this.side = side;
    }

    @Override
    public double area() {
        return side * side * 6;
    }

    @Override
    public double volume() {
        return side * side * side;
    }

    @Override
    public double perimeter() {
        return side * 12;
    }

    @Override
    public PositionedPolyhedron at(Point o) {
        // 1.d: questa anonymous class NON può essere scritta come una lambda,
        // perché ha 2 metodi
        return new PositionedPolyhedron() {
            @Override
            public Point origin() {
                return o;
            }

            @Override
```

```
            public Iterator<Point> iterator() {
                final Point u = o.move(side, side, side);
                final Point[] ps = new Point[]{
                        o, o.move(side, 0., 0.), o.move(0., side, 0.), o.move(0., 0., side),
                        u, u.move(side, 0., 0.), u.move(0., side, 0.), u.move(0., 0., side),
                };
                return Arrays.asList(ps).iterator();
            }
        };
    }

}

// 1.e
public static class Sphere implements Solid {
    private double ray;     // raggio della sfera

    public Sphere(double ray) {
        this.ray = ray;
    }

    @Override
    public double area() {
        return 4. * Math.PI * ray * ray;
    }

    @Override
    public double volume() {
        return 4. / 3. * Math.PI * ray * ray * ray;
    }

    @Override
    public PositionedSolid at(Point o) {
        // questa anonymous class potrebbe essere scritta una come una lambda invece,
        // perché ha 1 metodo solo:
        // return () -> o;
        return new PositionedSolid() {
            @Override
            public Point origin() {
                return o;
            }
        };
    }
}

// 1.f
public static void main(String[] args) {
    Cube cube1 = new Cube(11.), cube2 = new Cube(23.);
    Sphere sphere1 = new Sphere(12.), sphere2 = new Sphere(35.);
    List<Solid> solids = List.of(cube1, cube2, sphere1, sphere2);
    List<Cube> cubes = List.of(cube1, cube2);
    List<Sphere> spheres = List.of(sphere1, sphere2);
    List<? extends Polyhedron> polys = cubes;

    // f.i: questa invocazione è legale perché Cube implementa Comparable<Solid>
    Collections.sort(cubes);
```

```
        // 1.f.ii
        Collections.sort(spheres, Solid.comparatorBy(Sphere::area));

        // 1.f.iii
        Comparator<Cube> cmpCube = Solid.comparatorBy(Cube::perimeter);
        Comparator<Solid> cmpSolid = Solid.comparatorBy(Solid::area);
        Comparator<Sphere> cmpSphere = Solid.comparatorBy(Sphere::perimeter);   // non compila
        Comparator<Solid> cmpSolid2 = Solid.comparatorBy(Cube::area);           // non compila
        Comparator<Polyhedron> cmpPoly = Solid.comparatorBy(Polyhedron::volume);
        Comparator<Sphere> cmpSphere2 = Solid.comparatorBy(Solid::area);
        Comparator<Polyhedron> cmpPoly2 = Solid.comparatorBy(Solid::volume);
        Comparator<Cube> cmpCube2 = Solid.comparatorBy(Polyhedron::perimeter);

        // 1.f.iv
        Collections.sort(solids, cmpCube2);     // non compila
        Collections.sort(cubes, cmpSolid);
        Collections.sort(spheres, cmpSphere2);
        Collections.sort(solids, cmpPoly2);     // non compila
        Collections.sort(cubes, cmpSolid2);
        Collections.sort(cubes, cmpPoly);
        Collections.sort(spheres, cmpPoly2);    // non compila
        Collections.sort(polys, cmpSolid);

        // 1.f.v
        Point o = new Point(1., -1.5, 2.);
        for (Polyhedron poly : polys) {
            for (Point p : poly.at(o)) {
                System.out.println(p);
            }
        }
    }
}
```

4.2.2 Esercizio 2

```
import java.util.*;

public class Es2 {


    // 2.a
    @FunctionalInterface
    public interface BiFunction<A, B, C> {
        C apply(A a, B b);
    }

    // 2.b
    public static class Pair<A, B> {
        public final A fst;
        public final B snd;

        public Pair(A a, B b) {
            this.fst = a;
            this.snd = b;
        }
    }

    // 2.c
```

```
    public static class Triple<A, B, C> extends Pair<A, B> {
        public final C trd;

        public Triple(A a, B b, C c) {
            super(a, b);
            this.trd = c;
        }
    }

    // 2.d
    public static <A, B, C> Iterator<Triple<A, B, C>> evalIterator(Iterator<Pair<A, B>> it, BiFunction<A, B,
        return new Iterator<>() {
            @Override
            public boolean hasNext() {
                return it.hasNext();
            }

            @Override
            public Triple<A, B, C> next() {
                Pair<A, B> p = it.next();
                return new Triple<>(p.fst, p.snd, f.apply(p.fst, p.snd));
            }
        };
    }

}
```

4.2.3 Esercizio 3

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class Es3 {

    // 3.a
    public static class FiboSequence implements Iterable<Integer> {

        private final int max;
        private final Map<Integer, Integer> cache = new HashMap<>();

        public FiboSequence(int max) {
            this.max = max;
        }

        @Override
        public Iterator<Integer> iterator() {
            return new Iterator<>() {
                private int i = 0;

                @Override
                public boolean hasNext() {
                    return i < max;
                }

                @Override
                public Integer next() {
                    return fib(i++);
```

```
            }

            private int fib(int n) {
                if (n < 2) return 1;
                else {
                    Integer x = cache.get(n);
                    if (x != null) return x;
                    else {
                        int r = fib(n - 1) + fib(n - 2);
                        cache.put(n, r);     // si commenti questo statement per
                                             // disabilitare la cache e vedere la differenza
                                             // enorme di tempi di calcolo
                        return r;
                    }
                }
            }
        };
    }
}

// si lanci questo main e si osservi quanto velocemente la macchina produce 100 numeri di fibonacci:
// se non ci fosse la cache sarebbe drammaticamente più lento a causa delle continue ricorsioni
public static void main(String[] args) {
    for (int n : new FiboSequence(100)) {
        System.out.println(n);
    }
}


// 3.b
public static void main__desugared() {
    // questo for senza il terzo statement è equivalente ad un while
    for (Iterator<Integer> it = new FiboSequence(100).iterator(); it.hasNext(); ) {
        int n = it.next();
        System.out.println(n);
    }
}


// 3.c
public static class GlobalFiboSequence implements Iterable<Integer> {

    private final int max;
    private final static Map<Integer, Integer> cache = new HashMap<>(); // è tutto uguale a
                                                                        // FiboSequence eccetto per
                                                                        //la cache che è statica

    public GlobalFiboSequence(int max) {
        this.max = max;
    }

    @Override
    public Iterator<Integer> iterator() {
        return new Iterator<>() {
            private int i = 0;

            @Override
```

```
                public boolean hasNext() {
                    return i < max;
                }

                @Override
                public Integer next() {
                    return fib(i++);
                }

                private int fib(int n) {
                    if (n < 2) return 1;
                    else {
                        Integer x = cache.get(n);
                        if (x != null) return x;
                        else {
                            int r = fib(n - 1) + fib(n - 2);
                            cache.put(n, r);     // si commenti questo statement per disabilitare la cache
                                                 // e vedere la differenza enorme di tempi di calcolo
                            return r;
                        }
                    }
                }
            };
        }
    }

}
```

4.2.4 Esercizio 4

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;

public class Es4 {

    // 4.a
    public static class FactThread extends Thread {

        private int n, result;

        public FactThread(int n) {
            this.n = n;
        }

        public int getResult() {
            return result;
        }

    }

    // 4.b
    public static Collection<FactThread> multiFact(Collection<Integer> ns) {
        Collection<FactThread> r = new ArrayList<>();
        for (int n : ns) {
            r.add(new FactThread(n));
        }
        return r;
```

```
    }

    // 4.c
    public static void main(String[] args) {
        for (FactThread t : multiFact(Arrays.asList(1, 2, 3, 4, 5, 6))) {
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(String.format("%s calculated %d", t, t.getResult()));
        }
    }

}
```

4.2.5 Esercizio 5

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Random;

public class Es5 {

    // 5.a.i
    @FunctionalInterface
    public interface Function<A, B> {
        B apply(A x);
    }

    // 5.a.ii
    public static <A, B> Iterator<B> mapIterator(Iterator<A> it, Function<A, B> f) {
        return new Iterator<>() {
            @Override
            public boolean hasNext() {
                return it.hasNext();
            }

            @Override
            public B next() {
                return f.apply(it.next());
            }
        };
    }

    // 5.b.i
    public static class Pair<A, B> {
        public final A fst;
        public final B snd;

        public Pair(A a, B b) {
            this.fst = a;
            this.snd = b;
        }
    }

    // 5.b.ii
```

```
    public static <A> Iterator<Pair<A, A>> pairIterator(Iterator<A> it) {
        return new Iterator<>() {
            A last;

            @Override
            public boolean hasNext() {
                if (it.hasNext()) last = it.next();
                return it.hasNext();
            }

            @Override
            public Pair<A, A> next() {
                return new Pair<>(last, it.next());
            }
        };
    }

    // 5.c
    public static void main(String[] args) {
        Random rnd = new Random();
        List<Integer> c = new ArrayList<>();
        final int size = Math.abs(rnd.nextInt() % 50) + 1;
        for (int i = 0; i < size; ++i)
            c.add(Math.abs(rnd.nextInt() % 1000));
        System.out.println(String.format("lista di cateti (size = %d):\n%s", c.size(), c));

        Iterator<Pair<Integer, Integer>> cateti = pairIterator(c.iterator());
        Iterator<Double> ipotenuse = mapIterator(cateti, (Pair<Integer, Integer> p) ->
                                        Math.sqrt(Math.pow(p.fst, 2) + Math.pow(p.snd, 2)));

        for (int cnt = 1; ipotenuse.hasNext(); ++cnt) {
            System.out.println(String.format("ipotenusa #%d: %g", cnt, ipotenuse.next()));
        }
    }

}
```

4.2.6 Esercizio 6

```
import java.util.*;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingDeque;

public class Es6 {

    public interface Pool<T, R> {
        void add(T x);
        R acquire() throws InterruptedException;
        void release(R x);
    }

    // 6.a
    public interface BasicPool<T> extends Pool<T, T> {
    }

    // 6.b
    public static class SimplePool<T> implements BasicPool<T> {
```

```
    private BlockingQueue<T> q = new LinkedBlockingDeque<>();

    @Override
    public void add(T x) {
        q.add(x);
    }

    @Override
    public T acquire() throws InterruptedException {
        return q.take();
    }

    @Override
    public void release(T x) {
        add(x);
    }
}

// 6.c + 6.d

public interface Resource<T> {
    T get();
    void release();
}

public static class AutoPool<T> implements Pool<T, Resource<T>> {

    private BlockingQueue<T> q = new LinkedBlockingDeque<>();

    public void add(T x) {
        q.add(x);
    }

    public Resource<T> acquire() throws InterruptedException {
        T r = q.take();
        return new Resource<>() {

            @Override
            public T get() {
                System.out.println(String.format("acquired: %s", r));
                return r;
            }

            @Override
            public void release() {
                System.out.println(String.format("released: %s", r));
                add(r);
            }

            @SuppressWarnings("deprecation")
            @Override
            public void finalize() {
                release();
            }
        };
    }
```

```
        @Override
        public void release(Resource<T> x) {
            x.release();
        }
    }

    public static void main(String[] args) {
        AutoPool<Integer> pool = new AutoPool<>();
        Random rnd = new Random();
        for (int i = 0; i < 5; ++i) {
            pool.add(i);
        }

        try {
            while (true) {
                Resource<Integer> r = pool.acquire();
                System.out.println("using " + r.get());
                Thread.sleep(Math.abs(rnd.nextInt() % 1000));
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}
```

4.2.7 Esercizio 7

```
import org.jetbrains.annotations.NotNull;
import org.jetbrains.annotations.Nullable;

import java.util.Iterator;
import java.util.Random;

public class Es7 {

    // 7.b
    public static class RandomSingleton {
        private RandomSingleton() {
        }

        @Nullable
        private static Random instance;

        @NotNull
        private static Random instance() {
            if (instance == null)
                instance = new Random();
            return instance;
        }

        // fare un setter per cambiare il seed del PRNG è uno dei modi possibili per riprodurre
        // il comportamento originale della classe Random del JDK. Normalmente, per avere un
        // seed specifico è necessario costruire un oggetto Random tramite l'apposito
        // costruttore. Nella nostra conversione a singleton rappresentiamo invece questo
        // aspetto come un setter che internamente reistanzia il singleton in maniera
        // trasparente all'utente.
        public void setSeed(int seed) {
```

```
            instance = new Random(seed);
        }

        public static int nextInt() {
            return instance().nextInt();
        }

        public static boolean nextBoolean() {
            return instance().nextBoolean();
        }

        public static double nextDouble() {
            return instance().nextDouble();
        }

    }

    // 7.c
    public static class RandomIterator implements Iterator<Integer> {
        private int len;

        public RandomIterator(int len) {
            this.len = len;
        }

        @Override
        public boolean hasNext() {
            return len > 0;
        }

        @Override
        public Integer next() {
            --len;
            return RandomSingleton.nextInt();
        }
    }

}
```

4.2.8 Esercizio 8

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class Es8 {

    public static <A extends Comparable<B>, B> int compareMany(Collection<A> a, Collection<B> b) {
        Iterator<A> ia = a.iterator();
        Iterator<B> ib = b.iterator();
        int r = 0;
        while (ia.hasNext() && ib.hasNext()) {
            A x = ia.next();
            B y = ib.next();
            if ((r = x.compareTo(y)) != 0) break;
        }
        return ia.hasNext() && !ib.hasNext() ? 1 : !ia.hasNext() && ib.hasNext() ? -1 : r;
    }
```

```
}
```

4.2.9 Esercizio 9

```
import org.jetbrains.annotations.NotNull;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Random;

public class Es9 {

    // 9.a
    @FunctionalInterface
    public interface Predicate<T> {
        boolean apply(T x);
    }

    // 9.b
    public interface Either<T> {
        T onSuccess(T x);
        void onFailure(T x) throws Exception;
    }

    // 9.c
    public static class SkippableArrayList<E> extends ArrayList<E> {
        @NotNull
        public Iterator<E> iterator(Predicate<E> p, Either<E> f) {
            Iterator<E> it = super.iterator();
            return new Iterator<E>() {
                @Override
                public boolean hasNext() {
                    return false;
                }

                @Override
                public E next() {
                    E e = it.next();
                    if (p.apply(e))
                        return f.onSuccess(e);
                    else {
                        try {
                            f.onFailure(e);
                        } catch (Exception ex) {
                            ex.printStackTrace();
                        }
                        return e;
                    }
                }
            };
        }
    }

    // 9.d
    public static void main(String[] args) {
        List<Integer> l = new ArrayList<>();
```

```
        SkippableArrayList<Integer> skl = new SkippableArrayList<>();
        Random rand = new Random();
        for (int i = 0; i < 23; ++i)
            rand.nextInt(11);
        // invocazione con lambda come primo argomento
        skl.iterator((x) -> x > 5, new Either<>() {
            @Override
            public Integer onSuccess(Integer x) {
                return x + 1;
            }

            @Override
            public void onFailure(Integer x) throws Exception {
                l.add(x);
            }
        });

        // SINTASSI ALTERNATIVA: invocazione con anonymous class come primo argomento
        skl.iterator(new Predicate<>() {
        // IntelliJ suggerisce di convertire questa anonymous class in una lambda:
        // in tal caso riotterremo la sintassi di cui sopra
            @Override
            public boolean apply(Integer x) {
                return x > 5;
            }
        }, new Either<>() {
            @Override
            public Integer onSuccess(Integer x) {
                return x + 1;
            }

            @Override
            public void onFailure(Integer x) throws Exception {
                l.add(x);
            }
        });


    }

}
```

4.2.10 Esercizio 10

```
import java.util.ArrayList;
import java.util.Iterator;

public class Es10 {

    public static class FancyArrayList<E> extends ArrayList<E> {

        // 10.a
        @FunctionalInterface
        public interface Function<A, B> {
            B apply(A a);
        }

        // 10.b
```

```java
public Iterator<E> iterator(int step, Function<E, E> f) {
    return new Iterator<>() {
        private int pos = step > 0 ? 0 : size() - 1;

        @Override
        public boolean hasNext() {
            return pos >= 0 && pos < size();
        }

        @Override
        public E next() {
            E r = get(pos);
            pos += step;
            return f.apply(r);
        }
    };
}

// 10.c.i
@Override
public Iterator<E> iterator() {
    return iterator(1, (x) -> x);   // si può fare l'identità con la lambda (solo Java 8+)
}

// 10.c.ii
public Iterator<E> backwardIterator() {
    return iterator(-1, new Function<>() {  // oppure con una anonymous class semplicissima
        @Override
        public E apply(E x) {
            return x;
        }
    });
}

// 10.d
// bisogna chiamare questo metodo con un nome diverso da iterator()
// altrimenti il sorgente non compila
public Iterator<E> iterator__refactored(int step, Function<E, E> f) {
    return new FancyIterator<>(this, step, f);
}

// questa classe è il refactoring della anonynmous class dell'esercizio 10.b:
// la differenza è che conserva la enclosing instance esplicitamente
private static class FancyIterator<E> implements Iterator<E> {
    private FancyArrayList<E> enclosing;
    private Function<E, E> f;
    private int step, pos;

    public FancyIterator(FancyArrayList<E> parent, int step, Function<E, E> f) {
        this.enclosing = parent;
        this.step = step;
        this.f = f;
        pos = step > 0 ? 0 : parent.size() - 1;
    }

    @Override
    public boolean hasNext() {  // questo metodo è uguale
```

```
                return pos >= 0 && pos < enclosing.size();
            }

            @Override
            public E next() {
                E r = enclosing.get(pos);
                pos += step;
                return f.apply(r);
            }
        }

    }

}
```

4.2.11 Esercizio 11

```
import java.util.Comparator;
import java.util.List;

public class Es11 {

    // 11.a
    // versione con campi pubblici immutabili, ma si potevano fare anche i getter volendo
    public static class Pair<A, B> {
        public final A a;
        public final B b;
        public Pair(A a, B b) {
            this.a = a;
            this.b = b;
        }
    }

    // 11.b
    public static <E> Pair<E, E> findMinAndMax(List<? extends E> l, Comparator<E> c) {
        E min = l.get(0), max = min;
        for (E x : l) {
            if (c.compare(x, max) > 0) max = x;
            if (c.compare(x, min) < 0) min = x;
        }
        return new Pair<>(min, max);
    }

    // 11.c
    public static <E extends Comparable<E>> Pair<E, E> findMinAndMax(List<? extends E> l) {
        return findMinAndMax(l, new Comparator<>() {
            @Override
            public int compare(E a, E b) {
                return a.compareTo(b);
            }
        });
    }

}
```

4.2.12 Esercizio 12

```
public class Es12 {

    // tipi dati dal testo

    public interface Food {
        int getWeight();
    }

    public interface Animal extends Food {
        void eat(Food f);
    }

    public static class Mouse implements Animal {
        private int weight;

        public Mouse(int weight) {
            this.weight = weight;
        }

        @Override
        public void eat(Food f) {
            weight += f.getWeight() / 3;
        }

        @Override
        public int getWeight() {
            return weight;
        }
    }

    public static class Cat implements Animal {
        private int weight;

        public Cat(int weight) {
            this.weight = weight;
        }

        @Override
        public void eat(Food f) {
            weight += f.getWeight() / 10;
        }

        @Override
        public int getWeight() {
            return weight;
        }
    }

    public static class Cheese implements Food {
        @Override
        public int getWeight() {
            return 300;
        }
    }
```

```
    // 12.a
    // le interfacce Food e Animal non serve rifattorizzarle, sono a posto così
    public abstract class AbstractAnimal implements Animal {
        private int weight, div;

        protected AbstractAnimal(int weight, int div) {
            this.weight = weight;
        }

        @Override
        public void eat(Food f) {
            weight += f.getWeight() / div;
        }

        @Override
        public int getWeight() {
            return weight;
        }
    }

    public class Cat extends AbstractAnimal {
        protected Cat(int weight, int div) {
            super(weight, 5);   // valori del tema A
        }

        public class Mouse extends AbstractAnimal {
            protected Mouse(int weight, int div) {
                super(weight, 2);
            }
        }


        // 12.b.i
        public static void main(String[] args) {
            Animal tom = new Cat(200);     // costanti del tema A
            Animal jerry = new Mouse(10);
            jerry.eat(new Cheese());
            jerry.eat(new Food() {
                @Override
                public int getWeight() {
                    return 10;
                }
            });
            tom.eat(jerry);
            System.out.println(String.format("Tom now weights %d", tom.getWeight())); //b.ii 208

        }
    }
```

4.2.13 Esercizio 13

```
import java.util.List;

public class Es13 {

    public class Geometry { // questa enclosing class non è necessaria,
                            // essendoci già la classe Es2, tuttavia il testo è dato così
```

```
    public static class Point {
        public final double x, y;

        public Point(double x, double y) {
            this.x = x;
            this.y = y;
        }
    }

    public static abstract class Line<S extends Line<S>> implements Comparable<S> {
        public abstract double length();

        @Override
        public int compareTo(S l) {
            return Double.compare(length(), l.length());
        }
    }

    public static class Segment extends Line<Segment> {
        private Point p1, p2;

        public Segment(Point p1, Point p2) {
            this.p1 = p1;
            this.p2 = p2;
        }

        // 13.a
        @Override
        public double length() {
            // implementazione della length() per i Segment
            return Math.sqrt(Math.pow(p1.x - p2.x, 2.) + Math.pow(p1.y - p2.y, 2.));
        }
    }

    public static class Polyline extends Line<Polyline> {
        private List<? extends Point> points;

        public Polyline(List<? extends Point> points) {
            this.points = points;
        }

        // 13.b
        @Override
        public double length() {
            // implementazione della length() per le Polyline tramite la length() dei Segment
            double r = 0.;
            for (int i = 0; i < points.size() - 1; ++i)
                r += new Segment(points.get(i), points.get(i + 1)).length();
            return r;
        }
    }
}

// 13.c.i
private static class Point3D extends Point {
    private final double z;
```

```
        public Point3D(double x, double y, double z) {
            super(x, y);
            this.z = z;
        }
    }

    public static void main(String[] args) {
        Point a = new Point3D(0., 0., 0.),
              b = new Point3D(2., 2., 2.),
              c = new Point3D(2., 0., 3.);
        Polyline abc = new Polyline(List.of(a, b, c)), acb = new Polyline(List.of(a, c, b));
        System.out.println(String.format("max length = %g", max(abc, acb).length()));
    }

    // 13.c.ii
    public static <T extends Comparable<T>> T max(T a, T b) {
        return a.compareTo(b) > 0 ? a : b;
    }

    // 13.c.iii
    // sqrt(2) + 2

}
```