Eserciziario di Programmazione ad Oggetti mod. 2

a.a. 2021/2022

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.

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 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 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.

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.

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)`	◯ true	◯ false	◯ non compila	◯ eccezione
b.	`alice.equals(alice)`	◯ true	◯ false	◯ non compila	◯ eccezione
c.	`null.equals(david)`	◯ true	◯ false	◯ non compila	◯ eccezione
d.	`alice.equals(david)`	◯ true	◯ false	◯ non compila	◯ eccezione
e.	`alice.equalsTo(morgan)`	◯ true	◯ false	◯ non compila	◯ eccezione
f.	`morgan.equals(morgan)`	◯ true	◯ false	◯ non compila	◯ eccezione
g.	`morgan.equals(madonna)`	◯ true	◯ false	◯ non compila	◯ eccezione
h.	`morgan.equals(david)`	◯ true	◯ false	◯ non compila	◯ eccezione
i.	`morgan.equalsTo(david)`	◯ true	◯ false	◯ non compila	◯ eccezione
j.	`david.equalsTo(morgan)`	◯ true	◯ false	◯ non compila	◯ eccezione
k.	`madonna.equals(3)`	◯ true	◯ false	◯ non compila	◯ eccezione
l.	`madonna.equalsTo("Madonna")`	◯ true	◯ false	◯ non compila	◯ eccezione

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 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`.

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.

4 Soluzioni

Sezione 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 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.

Sezione 3.1:

1. (a) false
   (b) true
   (c) non compila/eccezione[1]
   (d) false
   (e) false
   (f) true
   (g) false
   (h) false
   (i) non compila
   (j) false
   (k) false
   (l) non compila

Sezione 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 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`.

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