Programmazione ad Oggetti mod. 2

20/6/2019

Studente _____ Matricola _____

1. 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) [3 punti] 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`[1]
- `p1.equals(p2) == p1.equalsTo(p2)`

(b) [2 punti] 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:

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

```java
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 */ }
}

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) [3 punti] 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) [1 punti] 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) [2 punti] 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:

```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) [6 punti] (0.5 punti per espressione) 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:

`alice.equals(null)`	◯ true	◯ false	◯ non compila	◯ eccezione
`alice.equals(alice)`	◯ true	◯ false	◯ non compila	◯ eccezione
`null.equals(david)`	◯ true	◯ false	◯ non compila	◯ eccezione
`alice.equals(david)`	◯ true	◯ false	◯ non compila	◯ eccezione
`alice.equalsTo(morgan)`	◯ true	◯ false	◯ non compila	◯ eccezione
`morgan.equals(morgan)`	◯ true	◯ false	◯ non compila	◯ eccezione
`morgan.equals(madonna)`	◯ true	◯ false	◯ non compila	◯ eccezione
`morgan.equals(david)`	◯ true	◯ false	◯ non compila	◯ eccezione
`morgan.equalsTo(david)`	◯ true	◯ false	◯ non compila	◯ eccezione
`david.equalsTo(morgan)`	◯ true	◯ false	◯ non compila	◯ eccezione
`madonna.equals(3)`	◯ true	◯ false	◯ non compila	◯ eccezione
`madonna.equalsTo("Madonna")`	◯ true	◯ false	◯ non compila	◯ eccezione

(h) 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. [2 punti] 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. [2 punti] 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. [2 punti] 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`.

Total for Question 1: 23

2. [11 punti] 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.

Total for Question 2: 11