Programmazione Orientato agli Oggetti
Metodologie di Programmazione
Modellazione e Gestione della Conoscenza

Agosto 2024

In questo documento sono riportate una selezione significativa delle domande apparse nei compiti del modulo di *Programmazione Orientata agli Oggetti* dei corsi di *Metodologie di Programmazione* e *Modellazione e Gestione della Conoscenza*. Questo strumento deve essere inteso come un compendio allo studio. La soluzione delle domande deve essere ricercata nel materiale del corso o svolgendo esercizi di programmazione. Per ogni domanda, inoltre, oltre ad individuare la risposta corretta, ogni studentessa e studente dovrebbe essere in grado di **motivare correttamente la risposta**. Negli esami varianti delle domande saranno riproposte. Si consiglia, quindi, di non affidarsi alla sola **memoria** ma di dedicarsi ad una comprensione approfondita degli argomenti.

1 Interfacce, Classi ed Ereditarietà

1. Cosa si intendiamo quando parliamo di un oggetto *POJO*?
2. Descrivere i concetti di *tipo statico* e *tipo dinamico* di una variabile.
3. Descrivere il ruolo dei metodi equals, toString e hashcode definiti nella classe Object ed il loro *contratto* d'uso.
4. Supponiamo che ClasseB sia una *sottoclasse* di ClasseA. Quali sono le conseguenze nel passare, in Java, passare un ClasseB[] dove un ClasseA[] è atteso?
5. Consideriamo la dichiarazione delle seguenti classi Java:

```
class ClasseA {

        public void m1( ) {
                System.out.println("ClasseA->m1()");
                m2();
        }

        public void m2( ) {
                System.out.println("ClasseA->m2()");
        }

}

class ClasseB extends ClasseA {

        public void m2( ) {
                System.out.println("ClasseB->m2()");
        }

}
```

Consideriamo, inoltre, la seguente porzione di codice:

```
ClasseA c = new ClasseB();
c.m1();
```

Quale è il risultato della sua esecuzione?

- ◯ Viene stampato a video:

  ```
  ClasseA->m1()
  ClasseB->m2()
  ```

- ◯ Viene stampato a video:

```
ClasseA->m1()
ClasseA->m2()
```

◯ Il codice non viene eseguito a seguito di un errore di tipo.

6. Consideriamo la dichiarazione delle seguenti classi Java:

```
class ClasseA {

        public void m1( ) {
                System.out.println("ClasseA->m1()");
                m2();
        }

        private void m2( ) {
                System.out.println("ClasseA->m2()");
        }

}

class ClasseB extends ClasseA {

        public void m2( ) {
                System.out.println("ClasseB->m2()");
        }

}
```

Consideriamo, inoltre, la seguente porzione di codice:

```
ClasseA c = new ClasseB();
c.m1();
```

Quale è il risultato della sua esecuzione?

◯ Viene stampato a video:

```
ClasseA->m1()
ClasseB->m2()
```

◯ Viene stampato a video:

```
ClasseA->m1()
ClasseA->m2()
```

◯ Il codice non viene eseguito a seguito di un errore di tipo.

7. Sia ClasseB una sottoclasse di ClasseA. Consideriamo ora le seguenti definizioni:

```
class ClasseC {
        public void m( ClasseA a ) {
                System.out.println("ClasseC->m()");
        }
}

class ClasseD extends ClasseC {
        public void m( ClasseB b ) {
                System.out.println("ClasseD->m()");
        }
}
```

(a) Consideriamo inoltre la seguente porzione di codice:

```
ClasseA a = new ClasseB();
ClasseC c = new ClasseD();
c.m(a);
```

Quale è il risultato della sua esecuzione?

- ◯ Viene stampato a video:

  ClasseC->m()

- ◯ Viene stampato a video:

  ClasseD->m()

- ◯ Il codice non viene eseguito a seguito di un errore di tipo.

(b) Eseguendo, invece, la seguente porzione di codice:

```
ClasseB b = new ClasseB();
ClasseC c = new ClasseD();
c.m(b);
```

Quale risultato si otterrà?

- ◯ Viene stampato a video:

  ClasseC->m()

- ◯ Viene stampato a video:

  ClasseD->m()

- ◯ Il codice non viene eseguito a seguito di un errore di tipo.

2 Programmazione Generica

1. In Java i *generics* consentono di parametrizzare una classe rispetto ad un dato tipo T.
   (a) Che differenza c'è tra un parametro di tipo `List<Object>` ed uno di tipo `List<?>`?

(b) Quale garanzie abbiamo su una variabile dichiarata `List<? extends ClassA>`? (Dove ClassA è una classe definita nella nostra applicazione). Come possiamo usare tale variabile?

(c) Quale garanzie abbiamo su una variabile dichiarata `List<? super ClassA>`? (Dove ClassA è una classe definita nella nostra applicazione). Come possiamo usare tale variabile?

2. Descrivere il meccanismo della *type erasure* in Java.

3. Supponiamo che ClasseB sia una *sottoclasse* di ClasseA. Quali sono le conseguenze nel passare, in Java, passare un List<ClasseB> dove un List<ClasseA> è atteso?

4. Indicare differenze ed usi delle seguenti dichiarazioni generiche in Java con particolare riferimento al loro uso nella dichiarazione di metodi:

   - List<?>;
   - List<? extends T>;
   - List<? super T>;

5. Consideriamo la seguente definizione di interfaccia:

```
interface MyInterface {
        int findMin(List<Integer> list);
        int findMin(List<Double> list);
}
```

   Quale è il risultato della sua compilazione?

   ◯ Viene correttamente compilata senza nessun errore;

   ◯ Viene compilata ma con dei warning;

   ◯ Non viene compilata a causa di un errore.

6. Consideriamo la seguente dichiarazione:

```
public class Wrapper<T> {

        private T value;

        public Wrapper(T value) {
                this.value = value;
        }

        public T get() {
                return value;
        }
```

```
    public void set(T value) {
            this.value = value;
    }

}
```

(a) Supponiamo di considerare la seguente dichiarazione di variabile:

```
Wrapper<? extends Number> w1;
```

Quale potrebbe essere un possibile assegnamento valido per la variabile w1 che non sia null e che utilizzi il costruttore di Wrapper?

(b) Se consideriamo, invece, la seguente dichiarazione:

```
Wrapper<? super Number> w2;
```

Quale potrebbe essere un possibile assegnamento valido per la variabile w2 che non sia null e che utilizzi il costruttore di Wrapper?

(c) Se consideriamo la dichiarazione di Wrapper definita precedentemente, la seguente porzione di codice è corretta da un punto di vista di tipi?

```
Wrapper<? super Number> w3 = new Wrapper<>("test");
```

Il seguente assegnamento è corretto da un punto di vista di tipo?

```
String str = w3.get();
```

(d) Se consideriamo la dichiarazione di Wrapper definita precedentemente, la seguente porzione di codice è corretta da un punto di vista di tipi?

```
Wrapper<? super Number> w3 = new Wrapper<String>("test");
```

(e) C'è differenza tra le seguenti dichiarazioni?

```
Wrapper<? super Number> w3 = new Wrapper<>("test");
Wrapper<? super Number> w3 = new Wrapper<String>("test");
```

(f) Assumiamo di avere la variable seguente dichiarazione di variabile:

```
Wrapper<? extends Integer> w4;
```

Consideriamo, inoltre, le seguenti linee di codice:

```
w4.set(o1);
o2 = w4.get();
```

Di quali classi, se esistono, devono essere istanze o1 ed o2 perché non sia riportato alcun errore di tipo?

(g) Supponiamo, inoltre, di avere la variabile seguente dichiarazione di variabile:

```
Wrapper<? super Integer> w5;
```

Consideriamo, inoltre, le seguenti linee di codice:

```
w5.set(o1);
o2 = w5.get();
```

Di quali classi, se esistono, devono essere istanze o1 ed o2 perché non sia riportato alcun errore di tipo?

3 Principi SOLID

1. Descrivere, anche attraverso esempi, i principi SOLID e descrivere il loro ruolo nell'estendibilità e manuteniblità del codice. Per ogni principio elencare:
   - Il nome;
   - La definizione;
   - Almeno un esempio di applicazione.
2. Consideriamo le seguenti classi Java:

```
public class Rectangle {
            private final double height;

            private final double width;

            public Rectangle(double height, idoublent width) {
                    this.height = height;
                    this.width = width;
            }

            public double getArea() {
                    return width*height;
            }

            public double getPerimeter() {
                    return 2*(width*height);
            }
}

public class Square extends Rectangle {

    public Square(double edge) {
            super(edge, edge);
    }

}
```

Le classi Rectangle e Square soddisfano il principio di sostituzione di Liskov?

3. Fornire un esempio di violazione del *Principio di Sostituzione di Liskov*.
4. Descrivere il principio *Open Closed* indicando i vantaggi derivanti sull'estendibilità del codice.
5. Introdurre il concetto di *code smell* descrivendo **almeno tre esempi** e fornendo le possibili contromisure.

4 Costrutti di Programmazione

1. Descrivere il ruolo dei metodi di default nelle interfacce Java.
2. Descrivere le interfacce funzionali ed il loro uso con le *lambda expressions*. Elencare inoltre i diversi modi per usare i *riferimenti a metodi* spiegandone il diverso uso.
3. Descrivere le interfacce e le classi sealed e come queste possano essere usate in Java.
4. Descrivere la sintassi e l'uso dell'*espressione switch* in Java.
5. Descrivere il *pattern matching di tipo* in Java e le relazioni con le interfacce/classi sealed.
6. Descrivere le dichiarazioni di record in Java ed il loro uso.

5 Programmazione Concorrente

1. Descrivere il ruolo dei metodi wait, notify e notifyAll nella programmazione concorrente in Java mostrando un possibile esempio d'uso.
2. Descrivere le differenze tra notify e notifyAll fornendo almeno un esempio in cui i due costrutti hanno comportamenti diversi.
3. Descrivere il ruolo delle variabili volatile nella programmazione concorrente in Java e fornire un esempio del loro uso.
4. Quando un metodo Java può essere definito *thread-safe*?
5. Consideriamo il seguente fragmento di codice:

```
private static boolean done = false;

public static void main(String[] argv) {
        Runnable hellos = () -> {
                for( int i=0 ; i<1000 ; i++ ) {
                        System.out.println("Hello "+i);
                }
```

```
                done = true;
        };
        Runnable goodbyes = () -> {
                int i=0;
                while (!done) { i++; }
                System.out.println("Goodbye "+i);
        };
        ExecutorService executor = Executors.newCachedThreadPool();
        executor.execute(hellos);
        executor.execute(goodbyes);
}
```

Possiamo garantire che entrambi i thread eseguiti all'interno del main termineranno la loro computazione? In caso la risposta sia no, quali modifiche dovremmo apportare al codice?

6. Descrivere le differenze tra l'uso delle variabili volatile e le primitive wait, notify e notifyAll.

7. Descrivere le principali classi disponibili nel package concurrent di Java ed i meccanismi che mettono a disposizione per implementare meccanismi di sincronizzazione di alto livello. Discutere, inoltre, le differenze con le primitive base di sincronizzazione basate su wait, notify e notifyAll.

8. Consideriamo le due seguenti porzioni di codice (dove l è un'istanza della classe Lock):

```
syncrhonized (l) {
        // Blocco di Codice
}
```

```
l.lock();
//Blocco di Codice
l.unclock();
```

Possiamo considerarle *equivalenti*?

9. Descrivere le interfacce e le classi messe a disposizione da Java per il supporto alla *computazione asincrona*.

10. Descrivere l'utilizzo dei Future e CompletableFuture nella programmazione concorrente in Java fornendo **almeno un esempio** d'uso.

6 Esercizi di Programmazione

1. Definire l'interfaccia sealed Expression con le relative classi permits che permettono di rappresentare le espressioni numeriche mediante *oggetti POJO*.

Fornire l'implementazione del seguente metodo che presa un'espressione e, utilizzando switch e pattern matching di tipo, restituisce la sua valutazione come double:

```
public double eval(Expression e) {
        ...
}
```

2. Il *call-by-need* permette di valutare una espressione solo al momento che ne abbiamo veramente bisogno. Tale meccanismo può essere implementato in Java da una classe della seguente forma:

```
public class OnNeed<T> {

        private final Supplier<T> supplier;

        public OnNeed(Supplier<T> supplier) {
                this.supplier = supplier;
        }

        public T eval() {
                return supplier.get();
        }

}
```

Tale implementazione non è corretta per due ragioni:

1. supplier.get() viene invocato ad ogni invocazioni di eval;
2. il metodo eval non è *thread-safe*.

Fornire un'implementazione del metodo eval che risolve questi due problemi.