Programare Orientată pe Obiecte

Static și final; Singleton Design Pattern

Obiective

- Înțelegerea conceptului de static în contextul claselor și instanțelor
- Utilizarea keywords-urilor static și final din Java
- Folosirea design-pattern-ului Singleton

Cuvântul-cheie "final". Obiecte immutable

Variabilele declarate cu atributul `final` pot fi inițializate **o singură dată**. Observăm că astfel unei variabile de tip referință care are atributul `final` îi poate fi asignată o singură valoare (variabila poate puncta către un singur obiect). O încercare nouă de asignare a unei astfel de variabile va avea ca efect generarea unei erori la compilare.

Totuși, obiectul către care punctează o astfel de variabilă poate fi modificat intern, prin apeluri de metode sau acces la câmpuri.

Exemplu:

```
class Student {

    private final Group group;                      // a student can change the group he was assigned in
    private static final int UNIVERSITY_CODE = 15; // declaration of an int constant

    public Student(Group group) {
        // reference initialization; any other attempt to initialize it will be an error
        this.group = group;
    }
}
```

Dacă toate atributele unui obiect admit o unică inițializare, spunem că obiectul respectiv este `immutable`, în sensul că *nu putem schimba obiectul in sine (informatia pe care o stocheaza, de exemplu), ci doar referinta catre un alt obiect.*. Exemple de astfel de obiecte sunt instanțele claselor `String` și `Integer`. Odată create, prelucrările asupra lor (ex.: `toUpperCase()`) se fac prin **instantierea de noi obiecte** și nu prin alterarea obiectelor înseși.

Exemplu:

```
String s1 = "abc";

String s2 = s1.toUpperCase();     // s1 does not change; the method returns a reference to a new object which can be accessed using s2 variable
s1 = s1.toUpperCase(); // s1 is now a reference to a new object
```

Observăm că în acest exemplu am folosit un String literal. Literalii sunt păstrați într-un `String pool` pentru a limita memoria utilizată. Asta înseamnă că dacă mai declarăm un alt literal “abc”, nu se va mai aloca memorie pentru încă un String, ci vom primi o referință către s-ul inițial. În cazul în care folosim constructorul pentru String se aloca memorie pentru obiectul respectiv și primim o referință nouă. Pentru a evidentia concret cum functioneaza acest `String pool`, sa luam urmatorul exemplu:

```
String s1 = "a" + "bc";
String s2 = "ab" + "c";
```

În momentul în care compilatorul va încerca să aloce memorie pentru cele 2 obiecte, va observa că ele conțin, de fapt, aceeași informație. Prin urmare, va instanția un singur obiect, către care vor pointa ambele variabile, s1 și s2. Observați că această optimizare (de a reduce memoria) e posibilă datorită faptului că obiectele de tip String sunt **immutable**.

O intrebare legitimă este, așadar, cum putem compara două String-uri (ținând cont de faptul că avem referințele către ele, cum am arătat mai sus). Să urmărim codul de mai jos:

```
String a = "abc";
String b = "abc";
System.out.println(a == b);  // True

String c = new String("abc");
String d = new String("abc");
System.out.println(c == d);  // False
```

Operatorul "==" compară *referințele*. Dacă am fi vrut să comparăm șirurile în sine am fi folosit metoda `equals`. Același lucru este valabil și pentru oricare alt tip referință: operatorul "==" testează egalitatea *referințelor* (i.e. dacă cei doi operanzi sunt de fapt același obiect).

Dacă vrem să testăm “egalitatea” a două obiecte, se apelează metoda: `public boolean equals(Object obj)`.

Reţineţi semnătura acestei metode!

O consecinta a faptului ca obiectele de tip String sunt imutabile este determinata de faptul ca efectuarea de modificari succesive conduce la crearea unui numar foarte mare de obiecte in String pool.

```
public static String concatenareCuClasaString()    {
   String s = "Java";
   for (int i=0; i<10000; i++){
      t = t + "POO";
   }
   return t;
}
```

În acest caz, numărul de obiecte create în memorie este unul foarte mare. Dintre acestea doar cel rezultat la final este util. Pentru a preveni alocarea nejustificată a obiectelor de tip String care reprezintă pași intermediari în obținerea șirului dorit putem alege să folosim clasa StringBuilder creată special pentru a efectua operații pe șiruri de caractere.

```
public static String concatenareCuClasaStringBuilder(){
   StringBuilder sb = new StringBuilder("Java");
   for (int i=0; i<10000; i++){
        sb.append("POO");
   }
   return sb.toString();
}
```

Cuvântul cheie final poate fi folosit și în alt context decât cel prezentat anterior. De exemplu, aplicat unei clase împiedică o eventuală derivare a acestei clase prin moștenire.

```
final class ParentClass {
}

class ChildClass extends ParentClass {
    // eroare de compilare, clasa ParentClass nu poate fi extinsa
}
```

În mod similar, în cazul în care aplicăm cuvântul cheie final unei metode, acest lucru împiedică o eventuală suprascriere a acelei metode.

```
class ParentClass {
   public final void dontOverride() {
        System.out.println("You cannot override this method");
   }
}

class ChildClass extends ParentClass {
    public void dontOverride()                 // eroare de compilare, metoda dontOverride() din
        System.out.println("But I want to!");  // clasa parinte nu poate fi suprascrisa
    }
}
```

Cuvântul-cheie "static"

După cum am putut observa până acum, de fiecare dată când cream o instanță a unei clase, valorile câmpurilor din cadrul instanței sunt unice pentru aceasta și pot fi utilizate fără pericolul ca instațierile următoare să le modifice în mod implicit.

Să exemplificăm aceasta:

```
Student instance1 = new Student("Alice", 7);
Student instance2 = new Student("Bob", 6);
```

În urma acestor apeluri, `instance1` și `instance2` vor funcționa ca entități independente una de cealaltă, astfel că modificarea câmpului `nume` din `instance1` nu va avea nici un efect implicit și automat în `instance2`. Există însă posibilitatea ca uneori, anumite câmpuri din cadrul unei clase să aibă valori independente de instanțele acelei clase (cum este cazul câmpului `UNIVERSITY_CODE`), astfel că acestea nu trebuie memorate separat pentru fiecare instanță.

Aceste câmpuri se declară cu atributul **static** și au o locație unică în memorie, care nu depinde de obiectele create din clasa respectivă.

Pentru a accesa un câmp static al unei clase (presupunând că acesta nu are specificatorul `private`), se face referire la clasa din care provine, nu la vreo instanță. Același mecanism este disponibil și în cazul metodelor, așa cum putem vedea în continuare:

```
class ClassWithStatics {

    static String className = "Class With Static Members";
    private static boolean hasStaticFields = true;

    public static boolean getStaticFields() {
         return hasStaticFields;
    }
```

```
}

class Test {

    public static void main(String[] args) {
        System.out.println(ClassWithStatics.className);
        System.out.println(ClassWithStatics.getStaticFields());
    }
}
```

Pentru a observa utilitatea variabilelor statice, vom crea o clasa care ține un contor static ce numără câte instanțe a produs clasa în total.

```
class ClassWithStatics {

    static String className = "Class With Static Members";
    private static int instanceCount = 0;

    public ClassWithStatics(){
        instanceCount++;
    }

    public static int getInstanceCount() {
        return instanceCount;
    }
}

class Test {

    public static void main(String[] args) {
        System.out.println(ClassWithStatics.getInstanceCount());       // 0

        ClassWithStatics instance1 = new ClassWithStatics();
        ClassWithStatics instance2 = new ClassWithStatics();
        ClassWithStatics instance3 = new ClassWithStatics();

        System.out.println(ClassWithStatics.getInstanceCount());       // 3
    }
}
```

Deși am menționat anterior faptul că field-urile și metodele statice se accesează folosind sintaxa `<NUME_CLASA>.<NUME_METODA/FIELD>` acesta nu este singura abordare disponibilă în libajul Java. Pentru a referi o entitate statică ne putem folosi și de o instanța a clasei în care se află metodă/field-ul accesat.

În acest caz nu este relevant dacă tipul obiectului folosit este diferit de cel al referinței în care e stocat(i.e. avem o referință a clasei Animal care referă un obiect de tipul Dog). Pentru apelul unei metode statice se va lua în considerare numai tipul referinței, nu și cel al instanței pe care o referă

```
class ClassWithStatics {

    static String className = "Class With Static Members";
    private static boolean hasStaticFields = true;

    public static boolean getStaticFields() {
         return hasStaticFields;
    }
}

class Test {

    public static void main(String[] args) {
        ClassWithStatics instance = new ClassWithStatic();
        System.out.println(instance.className);
        System.out.println(instance.getStaticFields());
    }
}
```

Deși putem accesa o entitate statică folosind o referință, acest lucru este contraindicat. Field-urile și metodele statice aparțin clasei și nu ar trebui să fie in nici un fel dependențe de existența unei instanțe.

Pentru a facilita o initializare facilă a field-urilor statice pe care o clasa le deține, limbajul Java pune la dispoziție posibilitatea de a folosi blocuri statice de cod. Aceste blocuri de cod sunt executate atunci când clasa în cauza este încărcată de către mașina virtuală de java. Încărcarea unei clase se face în momentul în care aceaste este referita pentru prima dată in cod (se crează o instanță, se apelează o metodă statică etc.) În consecință, blocul static de cod se va execută întotdeauna înainte că un obiect să fie creat.

```
class TestStaticBlock {
    static int staticInt;
    int objectFieldInt;
    static {
        staticInt = 10;
        System.out.println("static block called ");
    }
}

class Main {
```

```
    public static void main(String args[]) {

        // Desi nu am creat nici o instanta a clasei TestStaticBlock
        // blocul static de cod este creat, iar output-ul comenzii va fi 10
        System.out.println(TestStaticBlock.staticInt);
    }
}
```

Singleton Pattern

Pattern-ul Singleton este utilizat pentru a restricționa numărul de instanțieri ale unei clase la un singur obiect, deci reprezintă o metodă de a folosi o singură instanță a unui obiect în aplicație.

Utilizari

Pattern-ul Singleton este util în următoarele cazuri:

- ca un subansamblu al altor pattern-uri:
  - impreună cu pattern-urile Abstract Factory, Builder, Prototype etc. De exemplu, în aplicație dorim un singur obiect factory pentru a crea obiecte de un anumit tip.
- în locul variabilelor globale. Singleton este preferat variabilelor globale deoarece, printre altele, nu poluează namespace-ul global cu variabile care nu sunt necesare.

Singleton este utilizat des în situații în care avem obiecte care trebuie accesate din mai multe locuri ale aplicației:

- obiecte de tip logger
- obiecte care reprezintă resurse partajate (conexiuni, sockets etc.)
- obiecte ce conțin configurații pentru aplicație
- pentru obiecte de tip *Factory*.

Exemple din API-ul Java: java.lang.Runtime [http://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html], java.awt.Toolkit [http://docs.oracle.com/javase/8/docs/api/java/awt/Toolkit.html]

Din punct de vedere al design-ului și testarii unei aplicații de multe ori se evită folosirea acestui pattern, în test-driven development fiind considerat un ***anti-pattern***. A avea un obiect Singleton a carei referință o folosim peste tot prin aplicație introduce multe dependențe între clase și îngreunează testarea individuală a acestora.

In general, codul care folosește stări globale este mai dificil de testat pentru că implică o cuplare mai strânsă a claselor, și împiedică izolarea unei componente și testarea ei individuală. Dacă o clasă testată folosește un obiect singleton, atunci trebuie testat și singleton-ul. Soluția este simularea *mock-up* a singleton-ului în teste. Încă o problemă a acestei cuplări mai strânse apare atunci când două teste depind unul de celălalt prin modificarea singleton-ului, deci trebuie impusă o anumită ordine a rulării testelor.

Încercați să nu folosiți în exces metode statice și componente Singleton.

Implementare

Aplicarea pattern-ului Singleton constă în implementarea unei metode ce permite crearea unei noi instanțe a clasei dacă aceasta nu există, și întoarcerea unei referințe către aceasta dacă există deja. În Java, pentru a asigura o singură instanțiere a clasei, constructorul trebuie să fie *private*, iar instanța să fie oferită printr-o metodă statică, publică.

În cazul unei implementări Singleton, clasa respectivă va fi instanțiată **lazy** (*lazy instantiation*), utilizând memoria doar în momentul în care acest lucru este necesar deoarece instanța se creează atunci când se apelează `getInstance()`, acest lucru putând fi un avantaj în unele cazuri, față de clasele non-singleton, pentru care se face *eager instantiation*, deci se alocă memorie încă de la început, chiar dacă instanța nu va fi folosită (mai multe detalii și exemplu în acest articol [http://www.javaworld.com/article/2077568/learn-java/java-tip-67--lazy-instantiation.html])

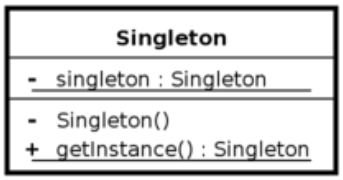

Fig. 1: Diagrama de clase pentru Singleton

Respectând cerințele pentru un singleton enunțate mai sus, în Java, putem implementa o componentă de acest tip în mai multe feluri, inclusiv folosind `enum`-uri în loc de clase. Atunci când îl implementâm trebuie avut în vedere contextul în care îl folosim, astfel încât să alegem o soluție care să funcționeze corect în toate situațiile ce pot apărea în aplicație (unele implementări au probleme atunci când sunt accesate din mai multe thread-uri sau când trebuie serializate).

```
public class Singleton {

  private static Singleton instance = null;

  private Singleton() {}

  public static Singleton getInstance() {
     if(instance == null) {
         instance = new Singleton();
     }
     return instance;
  }
  ...
}
```

- Instanța `instance` este *private*
- Constructorul este privat ca sa nu poata fi apelat decat din clasa respectivă
- Instanța este inițial nulă
- Instanța este creată la prima rulare a *getInstance()*

De ce Singleton și nu clase cu membri statici?

O clasă de tip Singleton poate fi extinsă, iar metodele ei suprascrise, însă într-o clasă cu metode statice acestea nu pot fi suprascrise (*overriden*) (o discuție pe aceasta temă puteți gasi aici [http://geekexplains.blogspot.ro/2008/06/can-you-override-static-methods-in-java.html], și o comparatie între static și dynamic binding aici [http://geekexplains.blogspot.ro/2008/06/dynamic-binding-vs-static-binding-in.html]).

Exerciții

1. (**3p**) Să se implementeze o clasă `PasswordMaker` ce generează, folosind `RandomStringGenerator`, o parolă pornind de la datele unei persoane. Această clasă o să conțină următoarele:
   - o constantă MAGIC_NUMBER având orice valoare doriți
   - un String constant MAGIC_STRING, lung de minim 20 de caractere, generat random
   - un constructor care primește: un String numit `name`
   - o metodă `getPassword()` care va returna parola
     - parola se construiește concatenand următoarele șiruri:
       - un șir random de lungime MAGIC_NUMBER, generat cu `RandomStringGenerator` și cu un alfabet obținut din 10 caractere obținute random din MAGIC_STRING
       - și șirul format prin conversia la String a lungimii lui name + un numar intreg generat random din intervalul [0,100]
   - Pentru subșiruri și alte metode utile consultați documentația clasei String [http://docs.oracle.com/javase/8/docs/api/java/lang/String.html]
2. (**3p**) Modificați implementarea clasei PasswordMaker astfel încât să respecte conceptul de **Singleton pattern** (să permită instanțierea unei singur obiect)
   - Pornind de la exemplul de Singleton din textul laboratorului implementați o versiune care urmează principiul de Eager Initialization (singura instanța a clasei este creată la pornirea aplicației, indiferent dacă este necesar sau nu)
   - Implementați o versiune de Singleton în care variabila `instance` este inițializata într-un bloc static
   - Adăugați un contor care să numere de câte ori a fost accesată metodă `getInstance()`. E nevoie ca acest contor să fie static?
   - *Tema de gândire:* Ce se va întâmplă dacă folosim conceptul de Singleton pattern într-un program paralelizat, care rulează pe mai multe linii de execuție (thread-uri). Ce probleme ar putea să apară?
3. (**3p**) Să se implementeze o clasă `MyImmutableArray` care să conțină:
   - un field de `ArrayList<Integer> immutableArray;` neinitializat în primă faza
   - un constructor care primește un ArrayList<Integer> și copiază toate elementele din acel array în `immutableArray`
   - o metodă getArray implementată în așa fel încât field-ul `immutableArray` să rămână immutable
4. (**1p**) Testați clasa `MyImmutableArray` demonstrând faptul că instanțele acestei clase sunt imutabile

Resurse

- Arhiva zip cu clasa RandomStringGenerator.java
