Laboratorul 4: static și final; Singleton pattern

Video introductiv: link [https://youtu.be/mjGOC6hmVH4]

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 cannot 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 (informația pe care o stochează, de exemplu), ci doar referința către un alt obiect.*. Exemple de astfel de obiecte sunt instanțele claselor `String` și `Integer`. Odată create, prelucrările asupra lor (ex.: `toUpperCase()`) se fac prin **instanțierea 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 alocă memorie pentru obiectul respectiv și primim o referință nouă. Pentru a evidenția concret cum funcționează acest `String pool`, să luăm următorul 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 întrebare 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 de 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 consecință a faptului că obiectele de tip String sunt imutabile este determinată de faptul că efectuarea de modificări succesive conduce la crearea unui număr foarte mare de obiecte in String pool.

```
public static String concatenationUsingTheStringClass() {
   String t = "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 concatenationUsingTheStringBuilderClass() {
   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 {
    // compilation error, the class ParentClass cannot be extended
}
```

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() {               // compilation error, the method dontOverride() from
        System.out.println("But I want to!");  // the parent class cannot be overriden
    }
}
```

Cuvântul-cheie "static"

După cum am putut observa până acum, de fiecare dată când creăm o instanță a unei clase, valorile câmpurilor din cadrul instanței sunt unice pentru aceasta și pot fi utilizate fără pericolul ca instanţierile 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>` aceasta nu este singura abordare disponibilă în limbajul Java. Pentru a referi o entitate statică ne putem folosi și de o instanță a clasei în care se află metoda/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 în nici un fel dependente de existența unei instanțe.

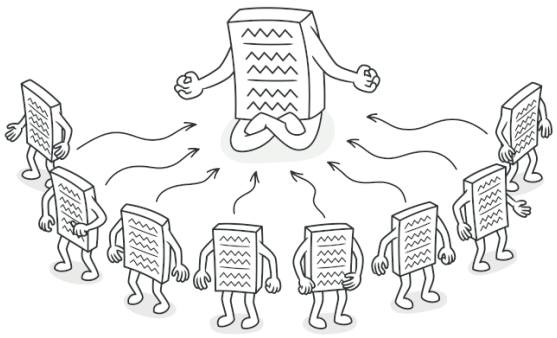
Pentru a facilita inițializarea field-urilor statice pe care o clasă le deține, limbajul Java pune la dispoziție posibilitatea de a folosi blocuri statice de cod. Aceste blocuri de cod sunt executate atunci când clasa în cauză este încărcată de către mașina virtuală de Java. Încărcarea unei clase se face în momentul în care aceasta este referită pentru prima dată în cod (se creează o instanță, se apelează o metodă statică etc.) În consecință, blocul static de cod se va executa 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[]) {
        // Even though we didn't create an instance of the TestStaticBlock class,
        // the static block of code is executed, and the output will be 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. Astfel, acest design pattern asigură crearea unei singure instanțe a clasei, oferind un punct de acces global al acesteia.

Cum ne asigurăm că o clasă are o singură instanță și că instanța este ușor accesibilă? O variabilă globală face un obiect accesibil, dar nu restricționeaza de la instanțierea mai multor obiecte.



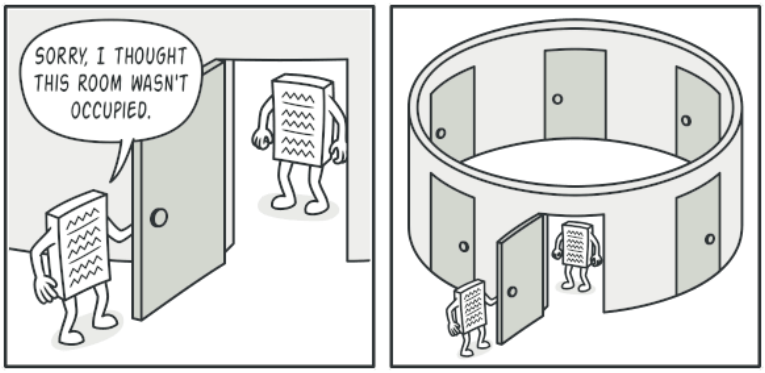
Utilizări

Pattern-ul Singleton este util în următoarele cazuri:

- ca un subansamblu al altor pattern-uri:
  - împreună 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 [https://docs.oracle.com/javase/7/docs/api/java/util/logging/Logger.html]
- obiecte care reprezintă resurse partajate (conexiuni, sockets [https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html] etc.)
- obiecte ce conțin configurații pentru aplicație
- pentru obiecte de tip *Factory* (acest design pattern va fi prezentat în cadrul laboratorului 9).



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 testării 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 cărei referință o folosim peste tot prin aplicație introduce multe dependențe între clase și îngreunează testarea individuală a acestora.

În 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.

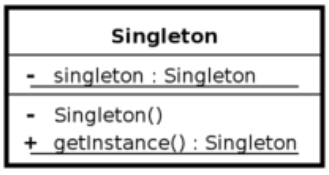
Încercați să nu folosiți în exces metode statice și componente Singleton.

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.

Lazy instantiation

În cazul unei implementări Singleton care folosește *lazy instantiation*, clasa respectivă va fi instanțiată **lazy**, 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])




```
public class SingletonLazy {
    private static SingletonLazy instance = null;

    private SingletonLazy() {}

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

- Instanța `instance` este *private*

- Constructorul este privat ca să nu poată fi apelat decât din clasa respectivă
- Instanța este inițial nulă
- Instanța este creată la prima rulare a *getInstance()*

Eager instantiation

În ceea ce privește instanțierea **eager**, instanța clasei este creată la momentul class loading [https://www.baeldung.com/java-classloaders], aceasta fiind cea mai ușoară metodă de a crea o clasă singleton. Are totusi un dezavantaj: instanța este creată chiar daca aceasta nu va fi folosita în cadrul aplicației.

```
public class SingletonEager {
    private final static SingletonEager instance = new SingletonEager();

    private SingletonEager() {}

    public static SingletonEager getInstance() {
        return instance;
    }
}
```

În cazul în care clasa Singleton nu folosește multe resurse, aceasta abordare este recomandata. În majoritatea scenariilor, clasele Singleton sunt create pentru resurse precum sistemul de fișiere, conexiunile la baza de date, iar instanțierea eager ar trebui evitată.

Static block instantiation

Implementarea este similară instanțierii *eager*, cu excepția faptului că instanța clasei este creată în blocul static care poate fi opțiune pentru gestionarea excepțiilor.

Atât inițializarea eager, cât și cea cu blocuri statice creează instanța înainte de a fi utilizată - motiv pentru care aceasta nu este cea mai bună practică de utilizat.

```
public class SingletonStaticBlock {
    private static SingletonStaticBlock instance = null;

    static {
        instance = new SingletonStaticBlock();
    }

    private SingletonStaticBlock() {}

    public static SingletonStaticBlock getInstance() {
        return instance;
    }
}
```

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 implementăm 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).

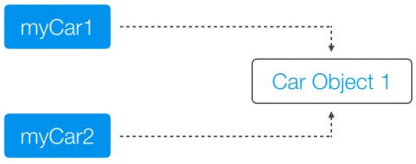
De ce Singleton și nu clase cu membri statici?

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 găsi aici [http://geekexplains.blogspot.ro/2008/06/can-you-override-static-methods-in-java.html], și o comparație între static și dynamic binding aici [http://geekexplains.blogspot.ro/2008/06/dynamic-binding-vs-static-binding-in.html]).

Deep copy. Shallow copy

Pentru început, ne propunem să ne familiarizăm cu noțiunea de copie în Java.

Noțiunea de *reference copy* implică copierea unei **referințe** care pointeaza către un obiect. Exemplu: Dacă avem un obiect de tipul Car, iar variabila myCar pointeaza către acesta, prin crearea unei copii de referință vom obține două variabile myCar ce pointeaza către același obiect de tipul Car.



Noțiunea de *object copy* creează însă o copie a obiectului în sine. Deci, dacă am copia din nou obiectul Car, am crea o copie a obiectului în sine, precum și o a doua variabilă de referință ce face referire la acel obiect copiat.

Shallow copy se referă la copierea obiectului „principal”, dar nu copiază obiectele “interioare”, acestea fiind “împărtășite” doar de obiectul original și copia acestuia. De exemplu, dacă pentru un obiect de tip Person, am crea un al doilea obiect Person, ambele obiecte ar avea aceleași obiecte Name și Address întrucât schimbarea unuia dintre acestea implică modificarea peste tot unde se regăsește referința obiectului modificat.

Putem spune astfel că cele două obiecte Person nu sunt independente - dacă este modificat obiectul Name al unui obiect Person, schimbarea se va reflecta și în celălat obiect de acest tip.

```
public class Person {
    private Name name;
    private Address address;

    public Person(Person firstPerson) {
         this.name = firstPerson.name;
         this.address = firstPerson.address;
    }
}
```

Spre deosebire de shallow copy, ***deep copy*** este o copie complet independentă a unui obiect. Dacă am copia obiectul Person, am copia întreaga structură a obiectului. În acest caz, o modificare a obiectului Adress nu va fi reflectată în cealalt obiect. Dacă aruncăm o privire asupra codului din exemplul următor, se observă că nu este folosit doar un *copy constructor* pentru obiectul Person, dar și *copy constructors* pentru obiectele interioare.

```
public class Person {
    private Name name;
    private Address address;

    public Person(Person otherPerson) {
         this.name    =  new Name(otherPerson.name);
         this.address =  new Address(otherPerson.address);
    }
}
```

Tutorial debugging în IntelliJ

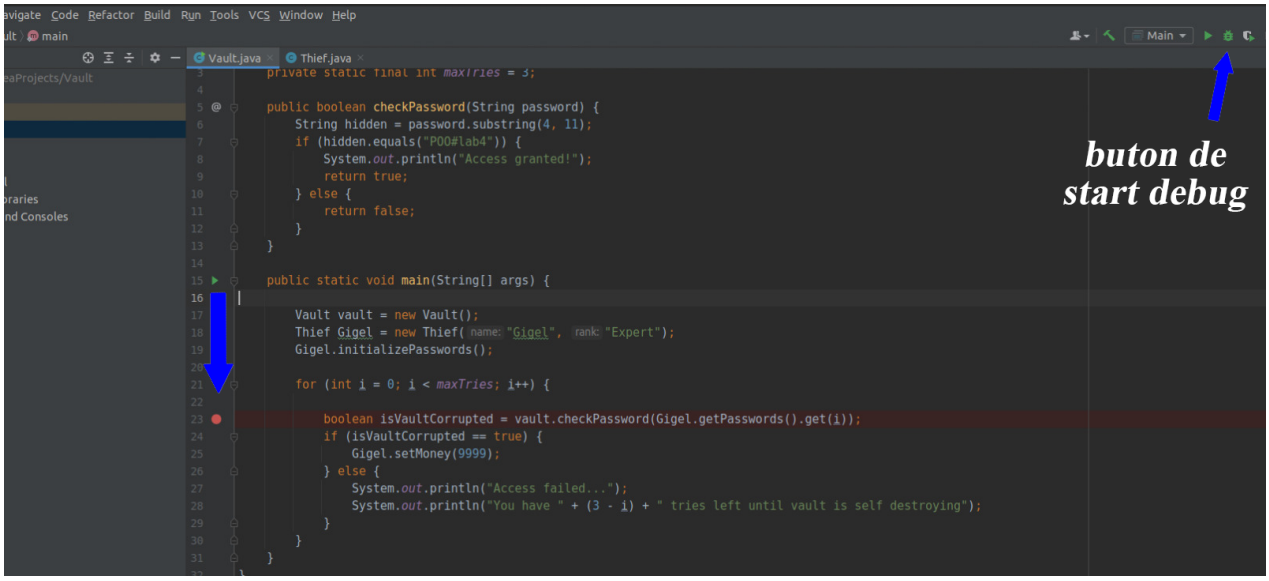
Luând în considerare următorul scenariu:

Gigel dorind să meargă la meci să îl vadă pe Messi jucând la PSG și pe Ronaldo la Man Utd, se decide că are nevoie de bani urgent și trebuie să spargă un seif. Acesta a făcut rost de 3 cartele având 3 parole diferite, știind că se poate folosi doar de acestea, chiar dacă unele sunt eronate sau nu. El a primit un pont că parola necesară seifului ar fi un subșir al unei cartele, care începe cu a 4-a literă și se termină cu a 11-a literă.

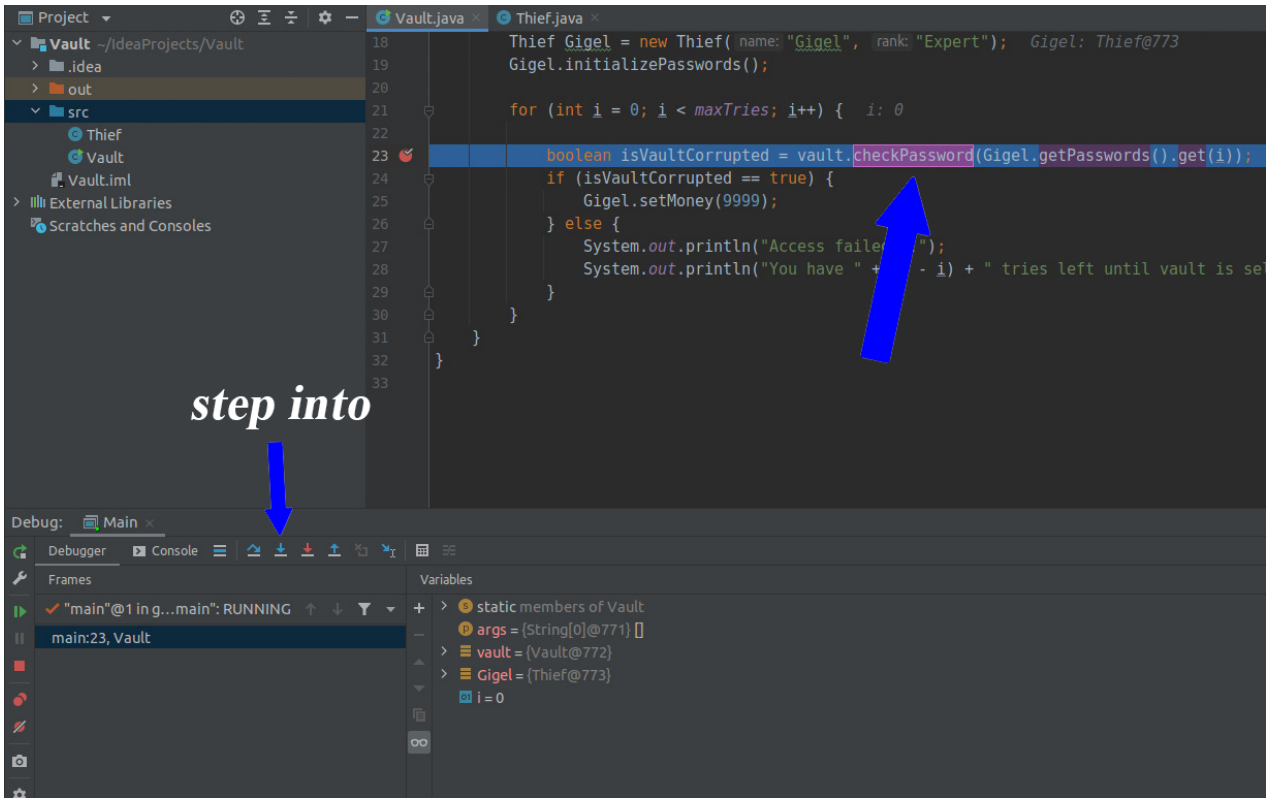
Mai întâi se uită la codul normal și vede că are doar 3 încercări, după care seiful se distruge. Nici una din parole nu funcționează, așa că acesta suspectează metoda `checkPassword` a seifului, deoarece până acolo nu se întâmplă nimic ciudat.

```
public boolean checkPassword(String password) {
                String hidden = password.substring(4, 11);
                if (hidden.equals("POO#lab4")) {
                        System.out.println("Access granted!");
                        return true;
                } else {
                        return false;
                }
        }
```

Așa că Gigel setează un **breakpoint** la linia 23 urmat de apăsarea pe butonul de **debug**.

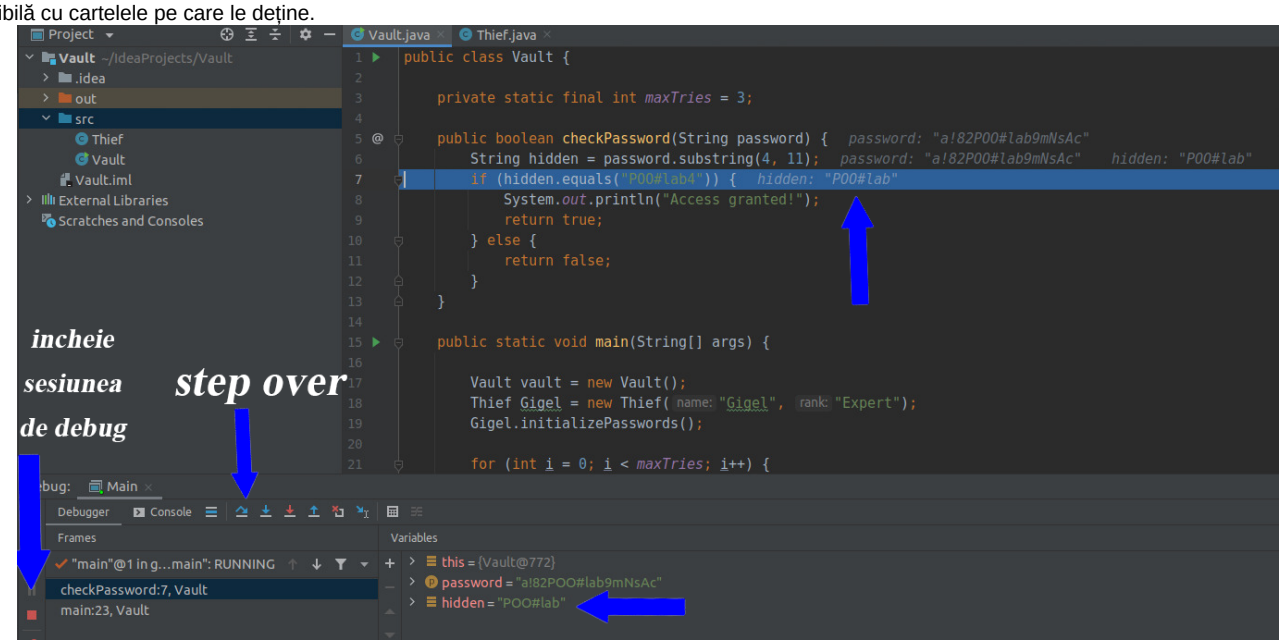


Apoi ca să inspecteze ce se întâmplă cu adevărat în metodă, apasă pe **Step Into**, urmat de un click pe metoda `checkPassword`.



Aici observă că `hidden` va stoca subșirul despre care a auzit în pontul său. Ca să vadă exact ce se va afla în `hidden` după executarea liniei 6, apasă pe **Step Over**.

În caseta de **Variables** sau direct în dreptul liniei 6 respectiv linia 7, Gigel observă că **hidden** conține doar `POO#lab`, iar intrarea în seif îi este imposibilă cu cartelele pe care le deține.



Acesta realizează că a primit un pont de la cineva care nu știa să folosească metoda `substring` și modifică codul sursă al seifului astfel încât să se preia subșirul de la a 4-a literă și să se oprească la cea de a 12-a literă.

```
String hidden = password.substring(4, 12); // schimbarea necesară
```

Acum, cartela sa care conținea înăuntru POO#lab4 va funcționa și va putea să își îndeplinească visul.

Extra

- în stânga - câmpul Frames arată stack-ul de apeluri
- se pot seta breakpoint-uri condiționale: setăm un breakpoint apoi click dreapta pe el și îi impunem o condiție care dacă se împlinește, debugger-ul va pune pauză execuției și vom putea face debug din acel punct
- Watch expression - în caseta de Variables apoi + se poate inițializa o variabilă nouă sau una deja existentă care să fie urmărită pe parcursul sesiunii de debug
- Set Value - putem modifica comportamentul programului fară să schimbăm codul sursă, pentru orice variabilă din caseta de Variables click dreapta → Set Value și putem observa cum se modifică programul cu noua valoare dată

Utilizarea clasei HashMap. Exemple

Clasa HashMap este un dicționar (tabelă de dispersie / hashtable), o colecție de perechi cheie-valoare unde avem o mapare între cheia unică în colecție și valoarea asociată acesteia, iar această clasă poate fi găsită în pachetul java.util. Principalele metode ale acestei clase sunt:

- `put(K key, V value)` - adaugă o pereche cheie-valoare sau înlocuiește valoarea asociată unei chei deja existente în HashMap.
- `get(Object key)` - accesează valoarea asociată unei chei din HashMap.
- `size()` - returnează numărul de elemente din HashMap.
- `containsKey(Object key)` - verifică dacă cheia există în HashMap.
- `remove(Object key)` - șterge o pereche cu cheia respectivă din HashMap.
- `clear()` - șterge toate elementele din HashMap.

Pentru a putea parcurge elementele unui HashMap se poate folosi for-each. Un exemplu de utilizare a metodelor clasei HashMap și de parcurgere a elementelor stocate de această clasă este următorul:

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

public class Main {
    public static void main(String[] args) {
        HashMap<String, Integer> grades = new HashMap<>();
        grades.put("Celentano", 5);
        grades.put("Firicel", 8);
        grades.put("Brinzoi", 9);
        grades.put("Bobita", 10);

        System.out.println("Firicel's grade: "+ grades.get("Firicel"));

        grades.put("Firicel", 10);
        System.out.println("Firicel's grade: "+ grades.get("Firicel"));

        for (Map.Entry<String, Integer> entry: grades.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }
    }
}
```

Exerciții

1. Pentru a rezolva laboratorul aveti urmatorul contest pe LambdaChecker : contest [https://beta.lambdachecker.io/contest/55/problems?page=1]
2. **(0 puncte)** Folosind pașii de mai sus și debugger-ul, reparați codul din pachetul `vault` din cadrul scheletului laboratorului [https://github.com/oop-pub/oop-labs/tree/master/src/lab4] (același care este prezentat în cadrul tutorialului de debug).
3. Aveți de implementat o mini aplicație de tip queries dintr-o bază de date cu dealershipuri și selleri de masini. Fiecare dealership are o colecție de asocieri dintre brandurile de masini vandute și pretul lor, de tip dicționar, iar fiecare seller are o listă de branduri de masina pe care le vinde.
    a. **(2 puncte)** Având la bază scheletul de cod, faceți clasa Database, care se ocupă cu gestionarea bazei de date a dealershipurilor și a sellerilor, o clasă de tip Singleton cu implementare de tip lazy. Această clasă va conține o listă de dealershipuri și o listă de selleri. Faceți aceste liste de tip final.
    b. **(1 punct)** Clasa `Dealership` are doua câmpuri: nume și un dicționar unde sunt stocate brandurile de masini, fiecare brand fiind asociat cu pretul sau la dealershipul respectiv. În această clasă, implementați următoarele:
        I. metoda `averagePrice`, care calculează media pretului masinilor din Dealership.
        II. copy constructorul clasei
        III. metoda `getPriceForBrand`, care primește ca parametru de intrare numele unui brand și întoarce pretul acestuia.
    c. **(1 punct)** Clasa `Seller` are doua câmpuri: nume si lista brandurilor pe care acesta le vinde. În această clasa implementați copy constructor.
    d. **(2 puncte)** În clasa `Database` implementați următoarele metode:
        I. `getAllDealerships` - întoarce lista de delaershipuri
        II. `getAllSellers` - intoarce lista de selleri
        III. `getDealershipByBrand` - primește ca parametru numele unui brand și întoarce lista cu dealershipurile care detin brandul respectiv
        IV. `getSellerByBrand` - primește ca parametru numele unui brand și întoarce lista cu sellerii care vand brandul respectiv.
        V. `getDealershipsByAveragePrice` - întoarce lista cu dealershipuri sortate crescător în funcție de pretul lor mediu.
        VI. `getDealershipsByPriceForBrand` - primește ca parametru numele unui brand și întoarce lista cu dealershipurile care detin acel brand, sortate după pretul acestuia în ordine crescătoare.
    e. **(1 punct)** În clasa `Seller` implementați următoarele metode, care vor apela metodele corespunzătoare din clasa `Database`:
        I. `getAllSellers`
        II. `getAllDealerships`
        III. `getSellersByBrand`
        IV. `getDealershipsByBrand`
        V. `getDealershipsByAveragePrice`
        VI. `getDealershipsByPriceForBrand`
    f. **(2 puncte)** În clasa Dealership implementați următoarele metode, care vor apela metodele corespunzătoare din clasa `Database` (atenție, aici metodele trebuie să întoarcă rezultate imutabile, sub forma de deep copy, folosind modalitatea prezentată în laborator, folosind Collections.unmodifiableList() [https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html] pentru ca evita posibilitatea de a modifica rezultatele, care sunt read-only):
        I. `getAllSellers`
        II. `getAllDealerships`

III. `getSellersByBrand`
IV. `getDealershipsByBrand`
V. `getDealershipsByAveragePrice`
VI. `getDealershipsByPriceForBrand`

g. **(1 punct)** În clasa `Database`, adăugați un contor drept câmp al clasei, de tip static, care va număra instanțierile clasei în cadrul metodei `getDatabase`, unde se face instanțierea clasei. Implementați metoda `getNumberOfInstances()` din `Database`, care întoarce numărul de instanțieri.

Pentru sortarea unui ArrayList (despre care am discutat în cadrul laboratorului trecut, puteți folosi metoda sort() din cadrul clasei ArrayList:

```
ArrayList<String> animals = new ArrayList<>();
animals.add("Dog");
animals.add("Cat");
animals.add("Sheep");

animals.sort(new Comparator<String>() {
    @Override
        public int compare(String o1, String o2) {
            return o1.compareTo(o2);
        }
});
```

Pentru shallow copy și deep copy la ArrayList avem în felul următor:

- shallow copy:

```
  List<Student> shallowCopy = new ArrayList<>(list);
```

- deep copy:

```
  List<Student> deepCopy = new ArrayList<>();
  for (var student: list) {
      // folosind constructor cu deep copy
      deepCopy.add(new Student(student));
  }
  return deepCopy;
```

poo-ca-cd/laboratoare/static-final.txt · Last modified: 2023/10/29 16:03 by calin.precupetu