

Laboratorul 7: Overriding, overloading & Visitor pattern

Video introductiv: link [https://www.youtube.com/watch?v=_mfLYYInv6c]

Obiective

Scopul acestui laborator este aprofundarea noțiunilor de programare orientată pe obiecte întâlnite în laboratoarele precedente, prezentarea design pattern-ului Visitor și familiarizarea cu situațiile în care acesta este util de aplicat.

Overloading

Supraîncărcarea se referă la posibilitatea de a avea într-o clasă mai multe metode cu același nume, dar implementari diferite. În Java, compilatorul poate distinge între metode pe baza semnăturii lor, acesta fiind mecanismul din spatele supraîncărcării.

Semnătura (signature) unei metode constă în:

- numele metodei
- numărul și tipul parametrilor

Opțional, pe lângă semnătura metodei poate fi menționat și tipul excepțiilor ce pot fi aruncate din codul acesteia.

Tipul de return al unei metode nu face parte din semnătura acesteia. Din acest motiv simpla modificare a tipului de return al unei metode nu este suficientă pentru supraîncărcare. Ceea ce vom primi este o eroare de compilare.

```
public class TRex {  
    public void eat(Triceratops victim) {  
        System.out.println("Take 5 huge bites");  
    }  
  
    public boolean eat(Triceratops victim) {  
        boolean satisfaction = false;  
        if (victim.isJuicy()) {  
            System.out.println("Eat and be satisfied");  
            satisfaction = true;  
        }  
        return satisfaction;  
    }  
}  
  
// Error "Duplicate method eat(Triceratops)" in type TRex
```

Observăm de asemenea că la compilare nu se ține cont de numele dat parametrilor. Astfel modificarea acestuia din *victim* în *dino*, spre exemplu, nu constituie o supraîncărcare validă.

Mai jos avem un exemplu valid de supraîncărcare pentru metoda *eat*:

```

public class TRex {

    public void eat(Triceratops victim) {
        System.out.println("Take 5 huge bites");
    }

    public void eat(Dromaeosaurus victim) {                // parametru cu tip diferit
        System.out.println("Take 1 single bite");
    }

    public void eat(Human firstCourse, Human secondCourse) { // numar si tipuri diferite de parametrii
        System.out.println("No humans to eat at the time");
    }

    public int eat(Grass desert) {                          // parametru cu tip diferit, return type este irelevant
        System.out.println("Rather starve");
        return 0;
    }

    public static void main(String [] args) {
        TRex john = new TRex();

        john.eat(new Triceratops());                // "Take 5 huge bites"
        john.eat(new Dromaeosaurus());              // "Take 1 single bite"
        john.eat(new Human("Ana"), new Human("Andrei")); // "No humans to eat at the time"
        john.eat(new Grass());                      // "Rather starve"
    }
}

```

O clasă poate supraîncărca metodele moștenite. Constructorii pot fi supraîncărcați.

Supraîncărcarea are loc la compilare, motiv pentru care mai este numită și polimorfism static (*compile time polymorphism*). În aceasta fază compilatorul decide ce metodă este apelată pe baza tipului referinței și prin analiza numelui și a listei de parametri. La runtime, când este întâlnit apelul unei metode supraîncărcate, deja se știe unde este codul care trebuie executat.

Overriding

Suprascrierea se referă la redefinirea metodelor existente în clasa părinte de către clasa copil în vederea specializării acestora. Metodele din clasa părinte nu sunt modificate. Putem suprascrivi doar metodele vizibile pe lanțul de moștenire (public, protected). O metodă din clasa copil suprascrivi metoda din clasa părinte dacă are același tip de return și aceeași semnătură (nume și parametri).

Spre deosebire de supraîncărcare care face acest lucru la compilare, în cazul suprascrierii se determină ce metodă va fi apelată, în mod dinamic, **la runtime**. Explicația este că decizia se face pe baza tipului obiectului care apelează metoda, deci a instanței (cunoscută la runtime). Din acest motiv, suprascrierea este cunoscută și ca polimorfism dinamic (Runtime polymorphism). **Polimorfismul reprezintă abilitatea unei clase să se comporte ca o altă clasă de pe lanțul de moștenire, și de aceea conceptul de suprascrriere a metodelor este foarte strâns legat. Supraîncărcarea, fiind la compile-time, nu are legătură cu acest polimorfism dinamic.**

La apelarea unei metode suprascrise, Java se uită la tipul intern al obiectului pentru care este apelată metoda, NU la referință. Astfel dacă referința are tipul clasei părinte, dar tipul este al clasei copil, JVM va apela metoda din clasa copil.

Câteva **reguli pentru suprascrriere** sunt:

- metoda suprascrisă are același tip de return și semnătură ca metoda inițială
- putem avea un tip de return diferit de cel al metodei inițiale, atâta timp cât este un tip ce moștenește tipul de return al metodei inițiale
- specificatorul de access al metodei suprascrise nu poate fi mai restrictiv decât cel al metodei inițiale
- nu poate arunca mai multe excepții sau excepții mai generale, poate însă arunca mai puține sau mai particulare sau excepții unchecked (de runtime)
- metodele de tip `static` și `final` nu pot fi suprascrise
- constructorii nu pot fi suprascriși

În exemplul de mai jos, metodele *purr* și *getFeatures* au fost suprascrise de tipul *GrumpyCat*.

```
class CatFeatures { }
class GrumpyCatFeatures extends CatFeatures { }
class GrumpyFeatures { }

class Cat {

    public void purr() {
        System.out.println("purrrr");
    }

    public CatFeatures getFeatures() {
        System.out.println("Cat getFeatures");
        return new CatFeatures();
    }

    public final void die() {
        System.out.println("Dying! frown emoticon");
    }

}

class GrumpyCat extends Cat {
    @Override
    public void purr() {
        System.out.println("NO!");
    }

    @Override
    public GrumpyCatFeatures getFeatures() {
        System.out.println("Grumpy getFeatures");
        return new GrumpyCatFeatures();
    }

    // compiler would complain if you included @Override here
    // @Override
    // public void die() { } // Cannot override the final method from Cat

    public static void main(String [] args) {
        ArrayList<Cat> cats = new ArrayList<Cat>();
        cats.add(new Cat());
        cats.add(new GrumpyCat());

        for (Cat c : cats) {
            c.purr();
            c.die();
        }
    }
}
```

```

        c.getFeatures();
    }
}

```

Adnotarea (Annotation) @Override este complet opțională. Totuși este indicat să o includeți mereu când suprascrieți o metodă. Motivele sunt simple:

- Compilatorul vă va anunța printr-o eroare dacă ați greșit numele metodei sau tipul parametrilor și această nouă metodă nu suprascrie de fapt o metodă a părintelui
- Face codul vostru mai ușor de citit, pentru că devine evident când o metodă suprascrie o altă metodă

O metodă cu argumente de tip primitiv nu poate fi suprascrisă cu o metodă cu tip wrapper.

`public void doSmth(int x)` nu poate fi suprascrisă cu `public void doSmth(Integer x)`

Metoda cu argument de tip wrapper poate primi și null, însă cea cu tipul primitiv nu, de aceea, neputând să fie păstrată echivalența, nu este permisă aceasta suprascriere

`super`

În laboratorul de agregare și de moștenire am folosit cuvântul cheie **super** pentru a invoca un anumit constructor din clasa părinte dar și pentru a apela în mod explicit metoda din clasa părinte în cazul metodelor suprascrise.

Rescriem metoda `purr()` din clasa `GrumpyCat` astfel:

```

@Override
public void purr() {
    super.purr();
    System.out.println("NO!");
}

```

La apelul metodei pe o instanță a clasei `GrumpyCat` output-ul va fi:

```

purrrr
NO!

```

Visitor

Design pattern-urile reprezintă soluții generale și reutilizabile ale unei probleme comune în design-ul software. Un design pattern este o descriere a soluției sau un template ce poate fi aplicat pentru rezolvarea problemei, nu o bucată de cod ce poate fi aplicată direct. În general pattern-urile orientate pe obiect arată relațiile și interacțiunile dintre clase sau obiecte, fără a specifica însă forma finală a claselor sau a obiectelor implicate.

Design pattern-ul *Visitor* oferă o modalitate de a *separa un algoritm de structura pe care acesta operează*. Avantajul constă în faptul că putem adăuga noi posibilități de prelucrare a structurii, fără să o modificăm. Extrapolând, folosind *Visitor*, putem adăuga noi funcții care realizează prelucrări asupra unei familii de clase, fără a modifica efectiv structura claselor.

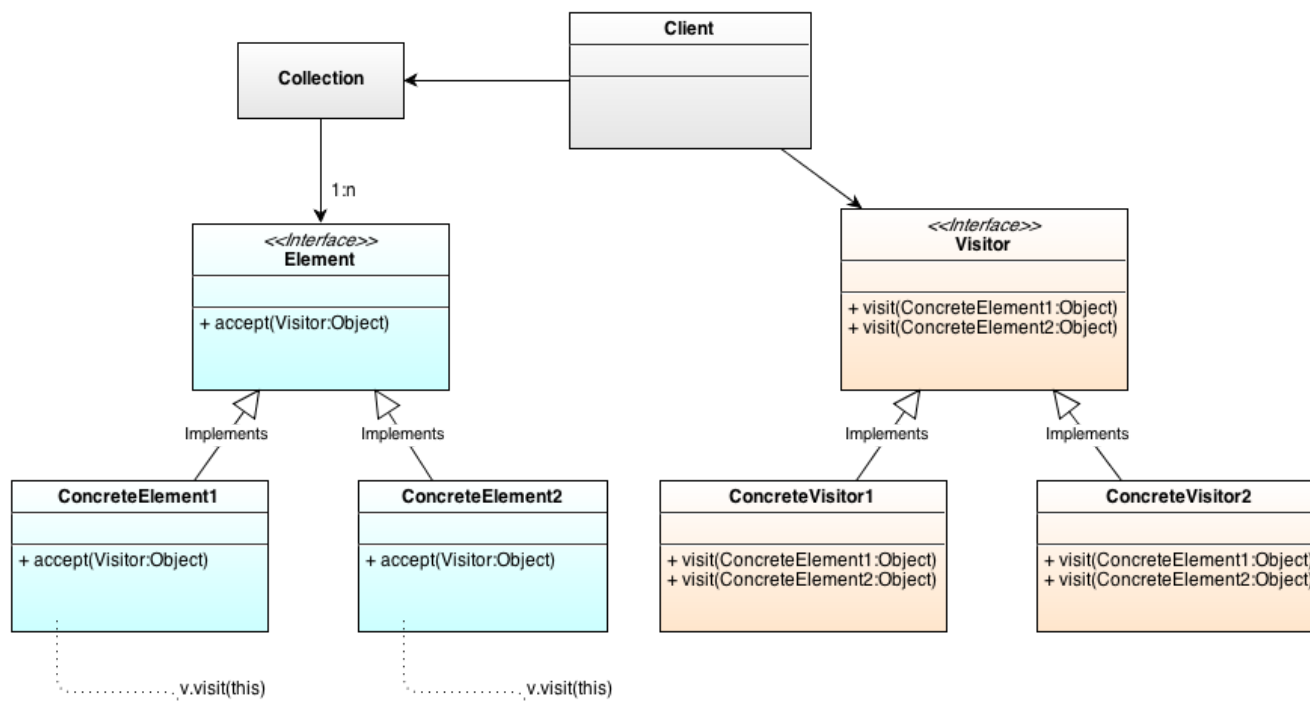
Acest pattern este comportamental (*behavioral*) pentru că definește modalități de comunicare între obiecte.

Cum recunoaștem o situație în care Visitor e aplicabil?

- Mai multe obiecte și operații pentru acestea
- Schimbarea/adăugarea operațiilor fără a modifica clasele
- Elemente heterogene - tipuri diferite de obiecte pe care se aplică mai multe operații

Decizia de utilizare a pattern-ului Visitor este în strânsă legătură cu stabilitatea ierarhiilor de clase prelucrate: dacă noi clase copil sunt adăugate rar, atunci se poate aplica acest pattern (într-o manieră eficientă), altfel nu este indicat.

Structură



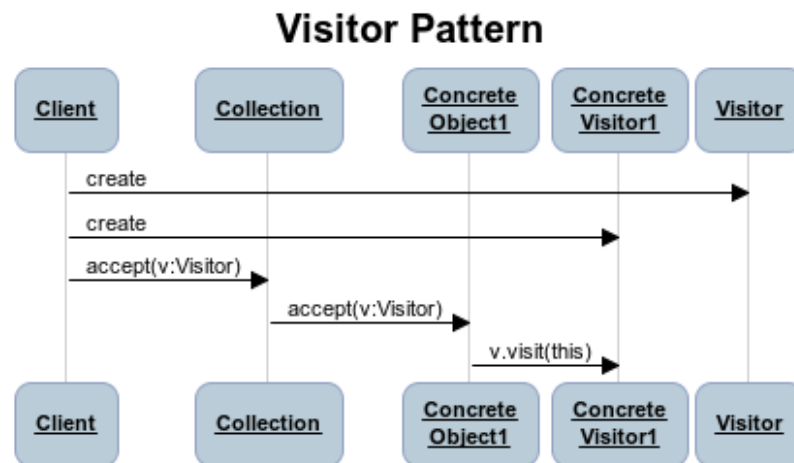
Visitor - o interfață pentru operația aplicată **Visitable** - o interfață pentru obiecte pe care pot fi aplicate operațiile (în diagramă este numită Element)

- metoda accept e independentă de tipul concret al Visitor-ului
- în accept se folosește obiectul de tip Visitor

Pentru fiecare algoritm/operație ce trebuie aplicată, se implementează clase de tip Visitor. În fiecare obiect de tip *Visitor* trebuie să implementăm metode care aplică operația pentru fiecare tip de element vizitabil.

În imaginea de mai jos este reprezentat **flow-ul aplicării acestui pattern**:

1. Clientul este cel care folosește o colecție de obiecte de unul sau mai multe tipuri, și dorește să aplice pe acestea diferite operații (în exercițiile din laborator clientul este practic programul vostru de test - main-ul). Clientul folosește obiecte *Visitor* create pentru fiecare operație necesară.
2. Clientul parcurge colecția și în loc să aplice operația direct pe fiecare obiect de tip *Element*, îi oferă acestuia un obiect de tip *Visitor*.
3. Obiectul de tip *Element* apelează metoda de "vizitare" oferită de *Visitor*.
4. Pe obiectul *Visitor* se apelează metoda *visit* corespunzătoare obiectului, iar în ea se efectuează operația. (⚠ în Visitor folosim conceptul de *overloading* pentru fiecare metodă *visit*)



Visitor și structurile de date

Aparent, folosirea lui *accept* este artificială. De ce nu declanșăm vizitarea unui obiect, apelând **direct** *v.visit(e)* atunci când dorim vizitarea unui obiect oarecare? Răspunsul vine însă chiar din situațiile în care vrem să folosim pattern-ul; vrem să lășăm structura internă a colecției să facă aplicarea vizitatorilor. Cu alte cuvinte vizitatorul se ocupă de fiecare obiect în parte, iar colecția îl "plimbă" prin elementele sale. De exemplu, când dorim să vizităm un arbore:

- declanșarea vizitării se va face printr-un apel *accept* pe un prim obiect (e.g. rădăcina arborelui)
- elementul curent este vizitat, prin apelul *v.visit(this)*
- pe lângă vizitarea elementului curent, este necesar să declanșăm vizitarea *tuturor elementelor accesibile din elementul curent* (e.g. nodurile-copil din arbore etc). Realizăm acest lucru apelând *accept* pe *fiecare* dintre aceste elemente. Acest comportament depinde de logica structurii.

Scenariu Visitor

Pentru a înțelege mai bine motivația din spatele design-pattern-ului *Visitor*, să considerăm următorul exemplu.

Before

Fie ierarhia de mai jos, ce definește un angajat (*Employee*) și un șef (*Manager*), văzut, de asemenea, ca un angajat:

Test.java

```

class Employee {
    String name;
    float salary;
    public Employee(String name, float salary) {
        this.name = name;
        this.salary = salary;
    }
    public String getName() {
        return name;
    }
    public float getSalary() {
        return salary;
    }
}
class Manager extends Employee {
    float bonus;
    public Manager(String name, float salary) {
        super(name, salary);
        bonus = 0;
    }
    public float getBonus() {
        return bonus;
    }
    public void setBonus(float bonus) {
        this.bonus = bonus;
    }
}
public class Test {
    public static void main(String[] args) {
        Manager manager;
        List<Employee> employees = new LinkedList<Employee>();
        employees.add(new Employee("Alice", 20));
        employees.add(manager= new Manager("Bob", 1000));
        manager.setBonus(100);
    }
}

```

Ne interesează să interogăm toți angajații noștri asupra *venitului lor total*. Observăm că:

- anagajații obișnuiți au salariul ca unic venit
- șefii posedă, pe lângă salariu, un posibil bonus

Varianta la îndemână ar fi să definim, în fiecare din cele doua clase, câte o metodă, *getTotalRevenue()*, care întoarce salariul pentru angajați, respectiv suma dintre salariu și bonus pentru șefi:

```

class Employee {
    ...
    public float getTotalRevenue() {
        return salary;
    }
}
class Manager extends Employee {
    ...

```

```

    public float getTotalRevenue() {
        return salary + bonus;
    }
}

```

Acum ne interesează să calculăm *procentul mediu* pe care îl reprezintă bonusul din venitul șefilor, luându-se în considerare doar bonusurile pozitive. Avem două posibilități:

- Definim câte o metodă, *getBonusPercentage()*, care în *Employee* întoarce mereu 0, iar în *Manager* raportul real. **Dezavantajul** constă în adăugarea în interfețe a unor funcții prea specializate, de detalii ce țin doar de unele implementări ale acestora.
- Parcurgem lista de angajați, *testăm*, la fiecare pas, tipul angajatului, folosind *instanceof*, și calculăm, doar pentru șefi, raportul solicitat. **Dezavantajul** este tratarea într-o manieră *neuniformă* a structurii noastre, cu evidențierea particularităților fiecărei clase.

Datorită acestor particularități (în cazul nostru, modalitățile de calcul al venitului, respectiv procentului mediu), constatăm că ar fi foarte utilă **izolarea implementărilor specifice** ale algoritmului (în cazul nostru, scrierea unei funcții în fiecare clasă). Acest lucru conduce, însă, la introducerea unei metode noi în *fiecare* din clasele antrenate în prelucrări, de fiecare dată când vrem să punem la dispoziție o nouă operație. Obținem următoarele dezavantaje:

- în cazul unui număr mare de operații, **interfețele claselor se aglomerează excesiv** și se ascunde funcționalitatea *de bază* a acestora
- codul din interiorul clasei (care servea funcționalității primare a acesteia) va fi amestecat cu cel necesar algoritmilor de prelucrare, devenind mai greu de parcurs și întreținut
- în cazul în care nu avem acces la codul claselor, singura modalitate de adăugare de funcționalitate este extinderea

În final, tragem concluzia că este de dorit să **izolăm algoritmii de clasele pe care le prelucrează**. O primă idee se referă la utilizarea *metodelor statice*. Dezavantajul acestora este că nu pot reține, într-un mod elegant, informație de stare din timpul prelucrării. De exemplu, dacă structura noastră ar fi arborescentă (recursivă), în sensul că o instanță *Manager* ar putea ține referințe la alte instanțe *Manager*, ce reprezintă șefii ierarhici inferiori, o funcție de prelucrare ar trebui să mențină o informație parțială de stare (precum suma procentelor calculate până într-un anumit moment) sub forma unor parametri furnizați apelului recursiv:

```

class Manager extends Employee {
    ...
    public float getPercentage(float sum, int n) {
        float f = bonus / getTotalRevenue();
        if (f > 0)
            return inferiorManager.getPercentage(sum + f, n + 1); // trimite mai departe cererea catre nivelul inferior
        return inferiorManager.getPercentage(sum, n);
    }
}

```

O abordare mai bună ar fi:

- conceperea claselor cu **posibilitatea de primire/atașare a unor obiecte-algoritm**, care definesc operațiile dorite
- definirea unor **clase algoritm** care vor vizita structura noastră de date, vor *efectua* prelucrările specifice fiecărei clase, având, totodată, *posibilitatea de încapsulare a unor informații de stare* (cum sunt suma și numărul din exemplul anterior)

After

Conform observațiilor precedente, structura programului Employee-Manager devine:

Test.java

```
interface Visitor {
    public void visit(Employee employee);
    public void visit(Manager manager);
}
interface Visitable {
    public void accept(Visitor v);
}
class Employee implements Visitable {
    ...
    public void accept(Visitor v) {
        v.visit(this);
    }
}
class Manager extends Employee {
    ...
    public void accept(Visitor v) {
        v.visit(this);
    }
}
public class Test {
    public static void main(String[] args) {
        ...
        Visitor v = new SomeVisitor();        // creeaza un obiect-vizitator concret
        for (Employee e : employees)
            e.accept(v);
    }
}
```

Iată cum poate arăta un vizitator ce determină venitul total al fiecărui angajat și îl afișează:

RevenueVisitor.java

```
public class RevenueVisitor implements Visitor {
    public void visit(Employee employee) {
        System.out.println(employee.getName() + " " + employee.getSalary());
    }
    public void visit(Manager manager) {
        System.out.println(manager.getName() + " " + (manager.getSalary() + manager.getBonus()));
    }
}
```

Secvențele de cod de mai sus definesc:

- o interfață, **Visitor**, ce reprezintă un *algorithm* oarecare, ce va putea vizita orice clasă. Observați definirea câte *unei metode visit(...)* pentru *fiecare clasă ce va putea fi vizitată*
- o interfață, **Visitable**, a carei metodă `accept(Visitor)` permite rularea unui *algorithm* pe structura curentă.
- implementări ale metodei `accept(Visitor)`, în cele două clase, care, pur și simplu, solicită vizitarea instanței curente de către vizitator.
- o implementare a unei operații aplicabilă pe obiectele de tip `Visitable`

În exemplul de mai sus, putem identifica :

- Element - Visitable
- ConcreteElement - Employee, Manager

Double-dispatch

Mecanismul din spatele pattern-ului Visitor poartă numele de **double-dispatch**. Acesta este un concept raspândit, și se referă la faptul că metoda apelată este determinată la *runtime* de doi factori. În exemplul Employee-Manager, efectul vizitarii, solicitate prin apelul `e.accept(v)`, depinde de:

- tipul elementului vizitat, *e* (*Employee* sau *Manager*), pe care se invocă metoda
- tipul vizitatorului, *v* (*RevenueVisitor*), care conține implementările metodelor *visit*

Acest lucru contrastează cu un simplu apel `e.getTotalRevenue()`, pentru care efectul este hotărât doar de tipul anagajatului. Acesta este un exemplu de **single-dispatch**.

Tutorialul de double-dispatch oferă mai multe detalii legate de acest mecanism.

Aplicabilitate

Pattern-ul **Visitor** este util când:


- se dorește prelucrarea unei *structuri complexe*, ce cuprinde mai multe obiecte de *tipuri diferite*
- se dorește definirea de *operații distincte pe aceeași structură*, pentru a preveni poluarea interfețelor claselor implicate, cu multe detalii aparținând unor algoritmi diferiți. În acest fel, se centralizează aspectele legate de același algoritm *într-un singur loc*, dar, în același timp, *se separă detaliile ce țin de algoritmi diferiți*. Acest lucru conduce la simplificarea atât a claselor prelucrate, cât și a vizitatorilor. Orice date specifice algoritmului rezidă în vizitator.
- **clasele ce se doresc prelucrate se modifică rar, în timp ce operațiile de prelucrare se definesc des**. Dacă însă sunt introduse multe clase vizitabile, după crearea obiectelor Visitor, atunci este necesară modificarea acestora din urmă, pentru adăugarea de metode *visit* pentru noile clase.

Avantaje:

- Decuplarea datelor de operațiile aplicate pe acestea
- Ușurează adăugarea unor noi operații/algoritmi. Se creează o implementare a unui obiect de tip Visitor și nu se schimbă nimic în obiecte vizitate.
- Spre deosebire de Iterator poate gestiona elemente de tipuri diferite
- Poate menține informații de stare pe măsură ce vizitează obiectele

Dezavantaje:

- Depinde de stabilitatea ierarhiei de obiecte vizitate. Adăugarea de obiecte vizitabile rezultă în schimbarea implementării obiectelor Visitor.

-  obiecte de noi tipuri adăugate des + multe operații aplicabile = NU folosiți Visitor
- Expune metode publice care folosesc informații de stare ale obiectelor. Nu se pot accesa membrii privați ai claselor, necesitatea expunerii acestor informații (în forma publică) ar putea conduce la *ruperea încapsulării*

Exemple din API-uri

Visitor este de obicei utilizat pentru structuri arborescente de obiecte:

- Parcurgerea arborilor de parsare
 - ASTVisitor [<https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FASTVisitor.html>] din Eclipse JDT. Folosind ASTParser [<http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FASTParser.html>] se creează arborele de parsare al codului dat ca intrare, iar ASTVisitor [<https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FASTVisitor.html>] parcurge arborele, oferind metode (*preVisit*, *postVisit*, *visit*) pentru multe tipuri de noduri (MethodDeclaration, Assignment, IfStatement etc.)
- Parcurgerea și vizitarea ierarhiei de directoare și fișiere
 - Java Nio - FileVisitor [<https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java.nio.file/FileVisitor.html>]
 - *FileVisitor* - interfața cu metode de vizitare
 - trebuie apelat Files.walkFileTree [[https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java.nio.file/Files.html#walkFileTree\(java.nio.file.Path,java.nio.file.FileVisitor\)](https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java.nio.file/Files.html#walkFileTree(java.nio.file.Path,java.nio.file.FileVisitor))] transmițându-i ca parametru un obiect care implementează *FileVisitor*
 - un tutorial [<http://docs.oracle.com/javase/tutorial/essential/io/walk.html>]

Summary

Supraîncărcarea (overloading) - mai multe metode cu același nume dar cu listă diferită de argumente

- metoda care va fi executată este stabilită la *compilare*, pe baza tipului referinței
- metoda supraîncărcată are neapărat o listă diferită de argumente și poate, opțional, avea:
 - alți modificatori de acces
 - alt tip de return
 - alte excepții
- constructorii pot fi supraîncărcați
- metodele moștenite pot fi supraîncărcate

Suprascrierea (overriding) - redefinirea metodelor moștenite

- metoda care va fi executată este stabilită la *runtime*, pe baza tipului obiectului

- metoda suprascrisă are același tip de return și semnătură ca metoda inițială
- putem avea un tip de return diferit de cel al metodei inițiale, atâta timp cât este un tip ce moștenește tipul de return al metodei inițiale
- specificatorul de access al metodei suprascrise nu poate fi mai restrictiv decât cel al metodei inițiale
- nu poate arunca mai multe excepții sau excepții mai generale, poate însă arunca mai puține sau mai particulare sau excepții unchecked (de runtime)
- metodele de tip `static` și `final` nu pot fi suprascrise
- constructorii nu pot fi suprascriși

Visitor - pattern pt modelarea comportamentului claselor

- util în situații în care:
 - avem mai multe obiecte și operații pentru acestea
 - dorim schimbarea/adăugarea operațiilor fără a modifica clasele
- indicat de utilizat pentru operații pe colecții și parcurgerea de structuri arborescente
- conceptul de double dispatch

Exerciții

Task 1 (8 puncte): Dorim să prelucrăm bucăți de text pe care să le convertim în diferite formate, momentan dokuwiki și markdown. Pentru un design decuplat între obiectele prelucrate și tipurile de formate dorite, implementați conversia folosind patternul Visitor.

- Fișierul **README** [<https://github.com/oop-pub/oop-labs/tree/master/src/lab7>] din scheletul de cod cuprinde informațiile necesare designului dorit.
 - implementați structura de clase din diagrama din README
 - implementați TODO-urile din scheletul de cod
- Pentru simplitatea testării scheletul oferă clasa `Test` care oferă bucățile de text pe care să le prelucrați.
 - dacă folosiți IntelliJ creați proiect din scheletul de laborator: File → New Project → select Java → select the skel folder
- În implementare va trebui să folosiți clasa `StringBuilder` [<https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/StringBuilder.html>]. Aceasta este o clasă mutabilă (*mutable*), spre deosebire de `String`, care e imutabilă (*immutable*). Vă recomandăm acest link [<https://www.geeksforgeeks.org/string-vs-stringbuilder-vs-stringbuffer-in-java/>] pentru un exemplu și explicații despre diferențele dintre ele.
- *Tips for faster coding:*
 - atunci când creați o clasă care implementează o interfață sau o clasă cu metode abstracte, nu scrieți de mână antetul fiecărei metode, ci folosiți-vă de IDE.
 - În IntelliJ va apărea cu roșu imediat după ce scrieți `extends...`/`implements...`. Dați `alt-enter` sau `option-enter` (pe mac), și vi se vor genera metodele pe care trebuie să le implementați, voi completând apoi conținutul lor.
 - generați constructorii folosind IDE-ul

Task 2 (2 puncte): Afișați numărul de fișiere cu extensia `".class"` sau `".java"` și numele acestora din directorul specificat în schelet.

- Pentru a rezolva exercițiul, veți implementa un FileVisitor [https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java.nio/file/FileVisitor.html], care extinde SimpleFileVisitor [https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java.nio/file/SimpleFileVisitor.html] din java.nio. Mai specific, veți suprascrie metoda visitFile [https://docs.oracle.com/javase/8/docs/api/java.nio/file/FileVisitor.html#visitFile-T-java.nio.file.attribute.BasicFileAttributes-], care este invocată atunci când un fișier este vizitat.
- Pentru mai multe detalii și un exemplu de implementare, consultați acest tutorial [http://docs.oracle.com/javase/tutorial/essential/io/walk.html] și scheletul de laborator.
- **Hint:** Folosiți `path.toString().endsWith` pentru a verifica extensia fișierelor. Dacă omiteți `toString`-ul nu o să funcționeze.
- **Hint:** Va trebui să verificați dacă instanța la care ați ajuns este un fișier sau nu, folosindu-vă de `BasicFileAttributes` [https://docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/BasicFileAttributes.html].

Task 3 (2 puncte): Copiați codul de la task-ul 1 și puneți-l în LambdaChecker [https://lambdachecker.io/problem/79], unde veți face testarea și submisia codului.

Referințe

1. Kathy Sierra, Bert Bates. *SCJP Sun Certified Programmer for Java™ 6 - Study Guide*. Chapter 2 - Object Orientation (available online [http://firozstar.tripod.com/_darksiderg.pdf]) - moștenire, polimorfism, overriding, overloading + exemple de întrebări
2. Vlissides, John, et al. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley (1995) (available online [http://index-of.co.uk/Software-Engineering/Design%20Patterns%20-%20Elements%20Of%20Reusable%20Object-Oriented%20Software%20-%20Addison%20Wesley.pdf])
3. Clasificarea design pattern-urilor [http://en.wikipedia.org/wiki/Software_design_pattern]
4. [Tutorial double-dispatch](#)

poo-ca-cd/laboratoare/visitor.txt · Last modified: 2021/11/22 08:54 by florin.mihalache