Laboratorul 7: Overriding, overloading & Visitor pattern

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

Obiective

- Implementarea polimorfismului în Java
- Diferența dintre Overriding & Overloading
- Prezentarea design pattern-ului Visitor și familiarizarea cu situațiile în care acesta este util de aplicat

Overriding

Suprascrierea se referă la redefinirea metodelor existente în clasa părinte de către clasa copil în vederea specializării acestora.

- Metodele din clasa parinte nu sunt modificate.
- Putem suprascrie doar metodele vizibile pe lanțul de moștenire (public, protected).
- O metodă din clasa copil suprascrie metoda din clasa părinte dacă are **același tip de return** și **aceeași semnatură**.

Semnătura (signature) unei metode constă în:

- numele metodei
- numărul și tipul parametrilor

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, care e 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 suprascriere a metodelor este foarte strâns legat.

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 suprascriere** sunt:

- metoda suprascrisă are același tip de return și semnatură ca metoda inițială
- putem avea un tip de return diferit de cel al metodei inițiale, atâta timp cat 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 si null, insă 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!
```

Overloading

Supraîncarcarea 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 supraîncărcarii.

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.

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

Spre deosebire de suprascriere, **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 întalnit apelul unei metode supraîncărcate, deja se știe unde este codul care trebuie executat.

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"
    }
}
```

Visitor Design Pattern

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 bucata de cod ce poate fi aplicata 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.

Visitor este un **behavioral design pattern** ce oferă posibilitatea de a adăuga în mod extern funcționalități pe o întreagă ierarhie de clase fără să fie nevoie să modificăm efectiv structura acestora.

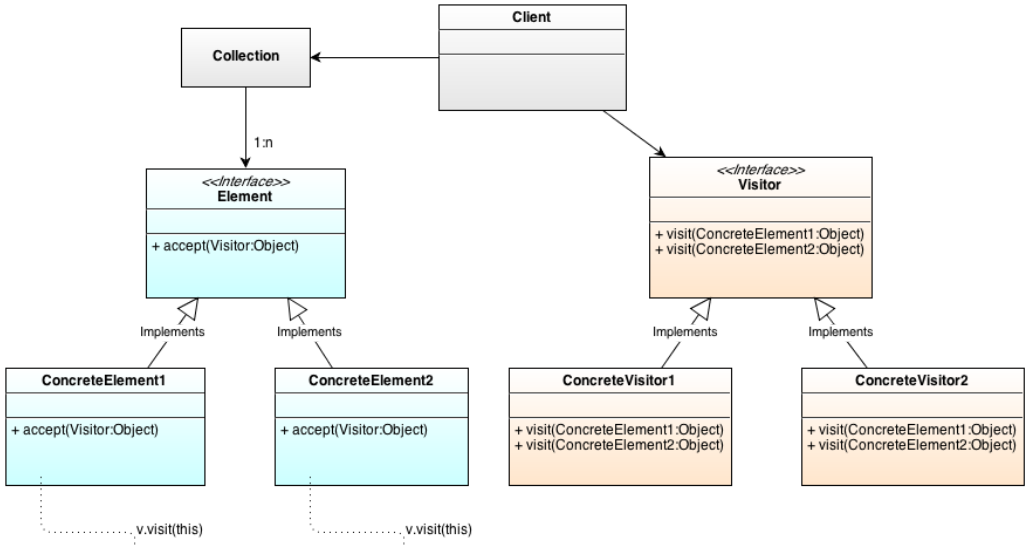
Acest pattern este behavioral (*comportamental*) pentru că definește modalități de comunicare între obiecte.

Aplicabilitate

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

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

Structură



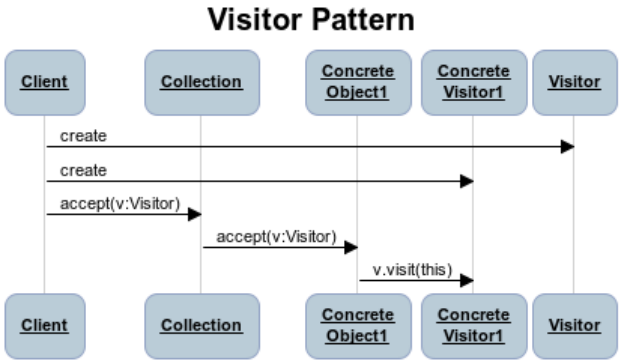
Visitor - o interfață pentru operația aplicată **Visitable** - o interfață pentru obiecte pe care pot fi aplicate operațiile (în diagramă este numită `Element`)

- 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 operaţia 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 declanşăm vizitarea unui obiect, apelând **direct** *v.visit(e)* atunci când dorim vizitarea unui obiect oarecare? Răspunsul vine însă chiar din situaţiile în care vrem să folosim pattern-ul; vrem să lăsăm structura internă a colecţiei să facă aplicarea vizitatorilor. Cu alte cuvinte vizitatorul se ocupă de fiecare obiect în parte, iar colecţia îl "plimbă" prin elementele sale. De exemplu, când dorim să vizităm un arbore:

- declanşarea vizitării se va face printr-un apel `accept` pe un prim obiect (e.g. rădacina arborelui)
- elementul curent este vizitat, prin apelul `v.visit(this)`
- pe lângă vizitarea elementului curent, este necesar sa declanşăm 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 înţelege mai bine motivaţia din spatele design-pattern-ului *Visitor*, să considerăm următorul exemplu.

Before

Fie ierarhia de mai jos, ce defineşte un angajat (*Employee*) şi un şef (*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 toţi angajaţii noştri asupra *venitului lor total*. Observăm că:

- anagajaţii obişnuiţi au salariul ca unic venit
- şefii 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 angajaţi, respectiv suma dintre salariu şi bonus pentru şefi:

```
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 şefilor, luându-se în considerare doar bonusurile pozitive. Avem două posibilităţi:

- Definim câte o metodă, *getBonusPercentage()*, care în *Employee* întoarce mereu 0, iar în *Manager* raportul real. **Dezavantajul** constă în adăugarea în interfeţe a unor funcţii prea specializate, de detalii ce ţin doar de unele implementări ale acestora.
- Parcurgem lista de angajaţi, *testăm*, la fiecare pas, tipul angajatului, folosind `instanceof`, şi calculăm, doar pentru şefi, raportul solicitat. **Dezavantajul** este tratarea într-o manieră *neuniformă* a structurii noastre, cu evidenţierea particularităţilor fiecărei clase.

Datorită acestor particularităţi (în cazul nostru, modalităţile de calcul al venitului, respectiv procentului mediu), constatăm că ar fi foarte utilă **izolarea implementărilor specifice** ale algoritmului (în cazul nostru, scrierea unei funcţii în fiecare clasă). Acest lucru conduce, însă, la introducerea unei metode noi în *fiecare* din clasele antrenate in prelucrări, de fiecare dată cand vrem să punem la dispoziţie o nouă operaţie. Obţinem următoarele dezavantaje:

- în cazul unui număr mare de operaţii, **interfeţele claselor se aglomerează excesiv** şi se ascunde funcţionalitatea *de bază* a acestora
- codul din interiorul clasei (care servea functionalităţii primare a acesteia) va fi amestecat cu cel necesar algoritmilor de prelucrare, devenind mai greu de parcurs şi întreţinut
- în cazul în care nu avem acces la codul claselor, singura modalitate de adăugare de funcţionalitate 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 reţine, într-un mod elegant, informaţie de stare din timpul prelucrării. De exemplu, dacă structura noastră ar fi arborescentă (recursivă), în sensul că o instanţă *Manager* ar putea ţine referinţe la alte instanţe *Manager*, ce reprezintă şefii ierarhic inferiori, o funcţie de prelucrare ar trebui să menţină o informaţie parţială de stare (precum suma procentelor calculate până într-un anumit moment) sub forma unor parametri furnizaţi 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/ataşare a unor obiecte-algoritm**, care definesc operaţiile 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 informaţii de stare* (cum sunt suma şi 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 şi îl afişează:

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()));
        }
}
```

Secvenţele de cod de mai sus definesc:

- o interfaţă, **Visitor**, ce reprezintă un *algoritm* oarecare, ce va putea vizita orice clasă. Observaţi definirea câte *unei metode visit(…)* pentru *fiecare clasă ce va putea fi vizitată*
- o interfaţă, **Visitable**, a carei metodă `accept(Visitor)` permite rularea unui algoritm pe structura curentă.
- implementări ale metodei `accept(Visitor)`, în cele două clase, care, pur şi simplu, solicită vizitarea instanţei 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, şi 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 conţine 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.

Cum implementăm?

1. Se declară interfața care să reprezinte elementul nostru, care va conține și metoda `public void accept(ElementVisitor elementVisitor);`
2. Se creează clasele concrete care implementează interfața declarată anterior. Body-ul pentru metoda accept va conține obligatoriu `elementVisitor.visit(this);`
3. Se definește o interfață care reprezintă Visitor-ul nostru și care va conține atâtea metode de visit câte clase concrete am creat la pasul anterior (câte o metodă de visit pentru fiecare tip de element).
4. Se creează o clasă concretă care implementează interfața de Visitor, unde vom adăuga implementările pentru fiecare tip de element în parte. (ex: `ElementDisplayVisitor()`)
5. În main putem testa dacă funcționează așa cum ne dorim iterând print-un arraylist/vector de obiecte de tip Element astfel: `elementIterator.accept(new ElementDisplayVisitor());`

Avantaje și dezavantaje

Avantaje:

- Decuplarea datelor de operațiile aplicate pe acestea
- Ușurează adăugarea unor noi operații/algortimi. 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 informaţii (in 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)] transmițându-i ca parametru un obiect care implementează *FileVisitor*
    - un tutorial [http://docs.oracle.com/javase/tutorial/essential/io/walk.html]

Summary

Overriding	Overloading
redefinirea metodelor moștenite	mai multe metode cu același nume, dar cu listă diferită de argumente
metoda care va fi executată este stabilită la **runtime**, pe baza **tipului obiectului**	metoda care va fi executată este stabilită la **compilare**, pe baza **tipului referinței**
POLIMORFISM DINAMIC	**POLIMORFISM STATIC**
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	metoda supraîncărcată are neapărat o listă diferită de argumente și poate, opțional, avea: • alți modificatori de acces • alt tip de return • alte excepții
constructorii nu pot fi suprascriși	constructorii pot fi supraîncărcati
metodele de tip static și final nu pot fi suprascrise	metodele moștenite pot fi supraîncărcate
putem folosi keyword-ul **super** pentru a apela în mod explicit metoda din clasa părinte în cazul metodelor suprascrise	-

Visitor = behavioral design pattern

- 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
- Folosește conceptul de double dispatch

Exerciţii

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 sa folositi 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 si explicații despre diferențele dintre ele.
- *Tips for faster coding*:
  - atunci cand creati o clasa care implementeaza o interfata sau o clasa cu metode abstracte, nu scrieti de mana antetul fiecarei metode, ci folositi-va de IDE.
  - In Intellij va aparea cu rosu imediat dupa ce scrieti extends…/implements… Dati alt-enter sau option-enter (pe mac), si vi se vor genera metodele pe care trebuie sa le implementati, voi completand apoi continutul lor.
  - generati constructorii folosind IDE-ul

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