***PAO – Curs 4***

Clase abstracte

Abstract class Angajat {

…

Public abstract double calculSalariu(); //nu are corp

…

}

Clasa abstracta :

* Date membere
* Constructori
* Metode ( abstracte sau nu )

Q: Cand folosim clase abstracte fara metode abstracte ? ( Interview Q ) – ex : HTTP Server

A: Cand in clasa sunt multe metode alternative pe care daca le-am declara pe toate cu abstract le-am pune cu corp vid

Angajat (abstracta)

Tablou cu elemente de mai multe tipuri

Angajati.txt

Ing -> 1, Popa Ion, …

Ec -> 2, Mihai Anca, ..

Economist

Inginer

Angajat []t = new Angajat[10]; -> nu o sa dea eroare pt ca t e de tip referinta si creeaza null-uri

t[0] = new Inginer ( ... ); // upcasting

t[1] = new Economist( ... ); // upcasting

for( ... ) {

x = t[i].calculSalariu();

...

}

t

Inginer

0 | 1 | 2 |…..| 9

Null|null|Null|null|Null|null

De ce as pune metode abstracte ?

Cand concep clasa consider indispensabile metodele pt utilizarea corecta a clasei.

Relatii de asociere intre clase/obiecte

Obiect de tip angajat cu componenta : |Persoana cu referinta catre un obiect extern Persoana : |”Pop Ion”|40 ani| Adresa| | 4000(salariu) | ...|

Avem relatie de tip asociere : ( Has\_A ).

Relatii de asociere:

* Agregare - durata de viata a obiectului referit ( incapsulat ) **nu** depinde de durata de viata a obiectului care il incapsuleaza
* Compozitia – altfel

Class CadruDidactic{

...Curs c;...

Public CadruDidactic(info cadru didactic, info curs){

...

c = new Curs(info curs); // Compozitie...}}

Class Facultate {

String nume, adresa;

... }

Class Curs {

String nume,..;

Facultate f; // Agregare ...

Public Curs (informatii despre curs, Facultate f){

...This.f = f; // Arata agregarea...}}

Q: Cand folosim extindere, cand folosim asociere(agregare/compozitie) ?

R: Cand se creeaza o ierarhie mare de clase, se aduna multe variabile redundante.

Ex: Sparg clasa mai mare in bucati mai mici si creed mai multe clase

Inginer -> InginerConstructor \_> InginerConstructorPoduri -> etc

In loc de a face extinderea de mai sus, fac :

Inginer | ConstructorPoduri -> cand vreau un inginer constructor poduri -> le combin

Siruri de caractere

Tabela de literali (String Pool)

String sir1 = “Java”;

String sir2 = “Java”;

a)Se creaza in tabela de literari o casuta cu “Java” si creeaza o referinta sir1 -> “java”.

La o declarare noua (sir 2) o sa verifice daca avem deja sirul “java” si v-a trimite referinta tot acolo .

b)Sir3 -> v-a tine o referinta catre “java” initializat in HEAP

**4 constructii :**

1. Cu tablou cu elemente de tip char ( dim. Fixa ) : char [t] = new char[100]
2. Clasa String
3. Clasa StringBuilder
4. Clasa StringBuffer

**Clasa String** :

String sir1 = „Java”;

String sir3 = new String ( „Java” ); // Cele doua declarari difera

1. -> economie de spatiu

Se poate muta (alocat cu new//dinamic ) **sir3 = sir3.intern();** -> sir 3 e mutat in tabela de literali si respecta regulile de acolo.

Clasa String este o ***clasa imutabila*** !! -> un obiect creat al acestei clase nu mai poate fi modificat ( mai exact, continutul nu poate fi modificat ). Se creaza un obiect nou de fiecare data

String sir1 = „java”;

Sir1.toUpperCase(); // se pierde valoarea creata „JAVA” in tabela.

System.out.println(sir1) = Java; // printeaza tot Java.

Corect -> sir1 = sir1.toUpperCase();

**String substring(int poz)**

subString (int p1, int p2) – inclusiv p1 si exclusiv p2 !!! -> ia valorile de la p1 pana la p2-1

interviu : daca avem String sir1 = „programare”;

cat este String sir2 = sir1.substring(2,5); -> sir2 = „ogr”

~~c = sir[1] ;~~ -> corect -> c= sir.charAt(1);

sir = sir1 + sir2 +sir3 +sir4; -> se concateneaza sir1 cu sir 2, obiectul temporar sir1,sir2 e concatenat cu sir3 -> temp2 + sir4->temp3 - Nu e eficient dpdv al memoriei.

**Clasa StringBuilder :**

If(sir1 == sir2)

...

Else

..

If(sir1.equils(sir2))

...

~~If(sir1==sir3)~~  // gresit pt ca una este in HEAP una este in tabela de literarli

Clasa mutabila! -> se poate modifica un sir fara se se mai creeze siruri temporare in memorie

StringBuilder sir = new StringBuilder(); -> v-a avea lungimea egala cu 16 caractere.

Aici sir.toUpperCase(); -> e modificat direct

Metode specifice : .append, .insert, .delete

Dezavantaj fata de String -> daca se modifica o valoare, se modifica pt toate -> **Nu e thread-safe!**

Solutie : **Clasa StringBuffer :**

Este StringBuilder, dar thread-safe ->  **Este mai lenta decat StringBuilder!**

**Ce folosesc? :** Tablou -> cazuri specifice ex:Criptografie | Urm : Sincronizare/Viteza. StringBuffer il folosim daca mai multe thread-uri folosesc acealsi string

Clase imutabile

Care sunt regulile de design pt o clasa de design :

* Toate datele membre nu trebuie sa fie modificate -> sunt final private
* Nu contine metode publice de tip set
* Clasa -> tip final ca sa nu fie extinsa. Daca vrem sa o extindem vrem sa avem toate **metodele finale**.
* Grija la ! Datele membre de tip referinta !
  + Daca avem o relatie de tip asociere folosim doar compozitie.
  + Metodele de tip get furnizeaza copii ale referintei.: tip referinta getDM(...) { return new(DM); } // returnam o copie