**Descrieti modurile in care este realizat bindingul in java (static, dinamic). Scrieti secvente de cod java pentru a exemplifica modul de definire si utilizarea acestuia. Explucati in 2-3 fraze caracteristicile pe care ati vrut sa le evidentiati.**

Binding-ul în Java poate fi de două tipuri: static (early) sau dynamic (late).

1. **Static Binding (Early Binding):**
   * Se realizează în timpul compilării.
   * Acesta este folosit pentru metodele private, finale și statice.
   * Nu se poate suprascrie.

**Exemplu de cod pentru static binding:**

class Animal {

void eat() {

System.out.println("Animal is eating");

}

}

public class Dog extends Animal {

void eat() {

System.out.println("Dog is eating");

}

public static void main(String args[]) {

Animal a = new Dog();

a.eat(); // Static binding, va apela metoda din clasa Animal

}

}

În acest exemplu, metoda **eat()** este legată la timpul de compilare, deoarece obiectul este de tipul **Animal**.

1. **Dynamic Binding (Late Binding):**
   * Se realizează în timpul execuției.
   * Este folosit pentru metodele care nu sunt statice, private sau finale.
   * Permite suprascrierea metodelor.

**Exemplu de cod pentru dynamic binding:**

class Animal {

void eat() {

System.out.println("Animal is eating");

}

}

public class Dog extends Animal {

void eat() {

System.out.println("Dog is eating");

}

public static void main(String args[]) {

Animal a = new Dog();

a.eat(); // Dynamic binding, va apela metoda din clasa Dog

}

}

În acest caz, tipul efectiv al obiectului **a** (care este **Dog**) determină ce metodă **eat()** va fi apelată, și acest lucru se decide la runtime.

Caracteristicile evidențiate:

* **Static Binding**: Este determinat la timpul compilării și nu permite suprascrierea, fiind limitat la tipul de referință.
* **Dynamic Binding**: Se bazează pe tipul obiectului la runtime, permițând polimorfismul și suprascrierea metodelor.

**Care dintre declaratii sunt corecte si care nu? argumentati raspunsul.**

**Collection<? super Integer> col1 = new Collection<Integer>();**

**List<? extends Number> col2 = new LinkedList<Double>();**

**List<? super Double> col3 = new ArrayList<Number>();**

**List<? extends Double> col4 = new ArrayList<Number>();**

**Collection<Integer> col5 = new List<Integer>();**

1. **Collection<? super Integer> col1 = new Collection<Integer>();**
   * **Incorect**: În primul rând, **Collection** este o interfață și nu poate fi instanțiată direct. În al doilea rând, **Collection<? super Integer>** înseamnă o colecție de obiecte de tipul Integer sau superclaselor sale (de exemplu, Number sau Object). O instanțiere cu **Collection<Integer>** nu respectă această regulă.
2. **List<? extends Number> col2 = new LinkedList<Double>();**
   * **Corect**: Această declarație este corectă. **List<? extends Number>** înseamnă o listă de obiecte care sunt fie de tipul Number, fie subtipuri ale acestuia. **Double** este un subtip al lui **Number**, deci **LinkedList<Double>** este valid.
3. **List<? super Double> col3 = new ArrayList<Number>();**
   * **Corect**: **List<? super Double>** înseamnă o listă de obiecte care sunt fie de tipul Double, fie superclaselor acestuia. **Number** este o superclasă a lui **Double**, deci **ArrayList<Number>** este un tip valid pentru această declarație.
4. **List<? extends Double> col4 = new ArrayList<Number>();**
   * **Incorect**: Această declarație nu este corectă deoarece **List<? extends Double>** sugerează o listă de obiecte care sunt Double sau subtipurile sale. Deoarece Double este o clasă finală (nu poate fi extinsă), singura opțiune validă aici ar fi **List<Double>**. **Number** nu este un subtip al lui **Double**.
5. **Collection<Integer> col5 = new List<Integer>();**
   * **Incorect**: Similar cu primul caz, **List** este o interfață și nu poate fi instanțiată direct. Trebuie să se utilizeze o clasă concretă care implementează interfața **List**, cum ar fi **ArrayList** sau **LinkedList**.

Prin urmare, declarațiile 2 și 3 sunt corecte, în timp ce declarațiile 1, 4 și 5 sunt incorecte.

**Compileaza urmatoarele programe in java? daca nu, de ce? daca da, ce afiseaza?Justificati in 2-3 randuri.**

1. **class Base {**

**int i;**

**Base() { add(1); }**

**void add(int v) { i += v; }**

**void print() { System.out.println(i); }**

**}**

**class Extension extends Base {**

**Extension() { add(2); }**

**void add(int v) { i += v \* 2; }**

**}**

**public class Test {**

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

**bogo(new Extension());**

**}**

**static void bogo(Base b) {**

**b.add(8); b.print();**

**} }**

**Compilează?** Da.

**Ce afișează?** Afișează "19".

**Justificare:**

* În **main**, se creează un obiect de tip **Extension**.
* În constructorul **Base**, se apelează **add(1)**. Din cauza polimorfismului, se va apela **add(int v)** din clasa **Extension**, adăugând **1\*2 = 2** la **i**.
* Constructorul **Extension** apelează **add(2)**, adăugând **2\*2 = 4** la **i** (acum **i = 6**).
* În **bogo**, se apelează **add(8)**, adăugând **8\*2 = 16** la **i** (acum **i = 22**).
* **b.print()** afișează valoarea lui **i**, care este 22.

**b) class Test {**

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

**B.C obj = new B().new C();**

**}**

**}**

**class A {**

**int val; A(int v) { val = v; }**

**}**

**class B extends A {**

**int val = 1; B() { super(2); }**

**class C extends A {**

**int val = 3;**

**C() {**

**super(4);**

**System.out.println(B.this.val);**

**System.out.println(C.this.val);**

**System.out.println(super.val);**

**} } }**

**class Test {**

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

**B.C obj = new B().new C();**

**}**

**}**

**class A {**

**int val;**

**A(int v) { val = v; }**

**}**

**class B extends A {**

**int val = 1;**

**B() { super(2); }**

**class C extends A {**

**int val = 3;**

**C() {**

**super(4);**

**System.out.println(B.this.val);**

**System.out.println(C.this.val);**

**System.out.println(super.val);**

**}**

**}**

**}**

**Compilează?** Da.

**Ce afișează?** Afișează "1", "3", "4".

**Justificare:**

* **B.C obj = new B().new C();** creează o instanță a clasei interne **C** din clasa **B**.
* În constructorul **C**, **super(4)** inițializează **val** din clasa **A** cu **4**.
* **System.out.println(B.this.val);** afișează valoarea **val** din instanța exterioară **B**, care este **1**.
* **System.out.println(C.this.val);** afișează valoarea **val** din instanța curentă **C**, care este **3**.
* **System.out.println(super.val);** afișează valoarea **val** din superclasă (**A**), care este **4**.

Top of Form

**Sa se defineasca o clasa comparabila Complex pentru un numar complex(re, im) cu valori intregi pentru partea reala si imaginara. Aceste valori nu sunt direct accesibile din alta clasa. criteriul natural de comparare este dupa partea reala, crescator si daca avem 2 valori egale atunci dupa cea imaginara, descrescator. Antetul clasei este: class Complex implements Comparable<Complex>. se vor defini:**

**- doi constructori, a) fara parametrii, initializare implicita cu 0, b) cu 2 parametrii - cate unul pentru fiecare data a clasei, avand aceiasi identificatori ca si datele clasei. Constr a) se def pe baza celuilalt constructor.**

**- metoda toString() + alte metode necesare pentru utilizarea datelor clasei.**

**- un main ce adauga intr-un obiect Vector 3 obiecte Complex, il sorteaza (ordine naturala) si ii afiseaza elmentele.**

Pentru a defini clasa **Complex** conform specificațiilor, vom urma pașii descriși mai jos:

1. **Definirea Clasei Complex**: Vom implementa interfața **Comparable<Complex>** pentru a defini criteriul de comparare.
2. **Constructori**:
   * Un constructor fără parametri care inițializează valorile cu 0.
   * Un constructor cu doi parametri pentru partea reală și cea imaginară.
3. **Metoda toString()** și orice alte metode necesare.
4. **Un main** pentru a demonstra utilizarea clasei.

Iată cum arată implementarea:

import java.util.Vector;

import java.util.Collections;

class Complex implements Comparable<Complex> {

private int re; // partea reala

private int im; // partea imaginara

// Constructor fara parametri

public Complex() {

this(0, 0);

}

// Constructor cu doi parametri

public Complex(int re, int im) {

this.re = re;

this.im = im;

}

// Metoda toString

public String toString() {

return "(" + re + ", " + im + "i)";

}

// Implementarea metodei compareTo

public int compareTo(Complex other) {

if (this.re == other.re) {

return other.im - this.im; // ordine descrescătoare pentru partea imaginara

}

return this.re - other.re; // ordine crescătoare pentru partea reală

}

// Metoda main pentru test

public static void main(String[] args) {

Vector<Complex> vector = new Vector<>();

vector.add(new Complex(1, 5));

vector.add(new Complex(1, 3));

vector.add(new Complex(2, 2));

Collections.sort(vector);

for (Complex complex : vector) {

System.out.println(complex);

}

}

}

**Explicații:**

* Constructorul fără parametri apelează constructorul cu parametri folosind **this(0, 0)**.
* **compareTo** este implementat pentru a respecta criteriul de ordonare specificat: mai întâi după partea reală (crescător), apoi după partea imaginară (descrescător).
* În metoda **main**, sunt create trei obiecte **Complex**, adăugate într-un **Vector**, sortate și apoi afișate.

Rulând **main**, veți obține o listă sortată a numerelor complexe conform criteriului specificat.

**Fie segmentul de cod. Ce linii pot fi decomentate, independent, fara a aparea erori la compilare? Argumentati!**

**class Big{ void met(int x){}}**

**class Heavy extends Big{**

**1. // void met(byte b) {}**

**2. /\* protected void met(int x) throws Exception {} \*/**

**3.// public void met(intx) {}**

**}**

**public class Weighty extends Heavy{**

**4.// voi met(int x) {}**

**5.//String met(int x) { return "hi";}**

**6.// public int met(intx) {return 7;}**

**7./\* private int met(char c) throws Error{ return 1;}**

**}**

Să analizăm fiecare linie de cod pentru a determina dacă poate fi decomentată fără a provoca erori la compilare. Trebuie să ne asigurăm că metodele suprascrise sau supraincarcate respectă regulile de suprascriere și supraincarcare din Java.

1. **// void met(byte b) {}**
   * **Poate fi decomentată**: Această linie definește o supraincarcare (overload) a metodei **met** în clasa **Heavy**. Deoarece are un tip diferit de parametru (**byte** în loc de **int**), aceasta este considerată o metodă diferită și nu va provoca erori la compilare.
2. **/\* protected void met(int x) throws Exception {} \*/**
   * **Nu poate fi decomentată**: Aceasta încearcă să suprascrie metoda **met** din clasa **Big**, dar reduce vizibilitatea metodei din **package-private** la **protected**, ceea ce nu este permis în Java. În plus, adăugarea unei noi excepții verificate (checked exception) la o metodă suprascrisă nu este permisă în Java.
3. **// public void met(intx) {}**
   * **Nu poate fi decomentată**: Acesta este un cod invalid deoarece lipsește un spațiu între tipul parametrului (**int**) și numele său (**x**). Dacă se corectează greșeala de sintaxă (de exemplu, **public void met(int x) {}**), aceasta poate fi decomentată fără probleme.
4. **// void met(int x) {}**
   * **Nu poate fi decomentată**: Aceasta încearcă să suprascrie metoda **met** din **Big**, dar reduce vizibilitatea metodei de la **public** (implicit în **Heavy** din cauza suprascrierii) la **package-private**, ceea ce nu este permis în Java.
5. **// String met(int x) { return "hi";}**
   * **Nu poate fi decomentată**: Deși numele și parametrii metodei sunt aceiași, tipul de retur diferă (**String** în loc de **void**). În Java, metodele suprascrise trebuie să aibă același tip de retur sau un tip de retur covariant. Aici, tipurile de retur sunt incompatibile, deci aceasta linie va provoca o eroare la compilare.
6. **// public int met(intx) {return 7;}**
   * **Nu poate fi decomentată**: Similar cu punctul 3, există o eroare de sintaxă. Dacă se corectează (de exemplu, **public int met(int x) {return 7;}**), acest cod va genera o eroare la compilare, deoarece tipul de retur este diferit de cel al metodei pe care o suprascrie (**void** în clasa părinte).
7. **/\* private int met(char c) throws Error{ return 1;} \*/**
   * **Poate fi decomentată**: Aceasta definește o nouă metodă în clasa **Weighty** care nu intră în conflict cu metoda **met** din clasele părinte, deoarece are parametri diferiți. Nu este o suprascriere, ci o supraincarcare, așa că nu vor apărea probleme la compilare.

Prin urmare, liniile care pot fi decomentate fără a provoca erori la compilare sunt liniile 1 și 7, cu mențiunea că în linia 3 trebuie corectată eroarea de sintaxă pentru a fi validă

**Compilează următoarele programe Java? Dacă da, ce afişează? Dacă nu, care ar fi rezolvarea? Justificați!**

**Subpunct a**

Să analizăm fiecare program în parte:

**Subpunctul a)**

class Hobbit {

int count(int x, int y) {

return x + y;

}

}

public class Frodo extends Hobbit {

public static void main(String[] args) {

Short myGold = 7;

System.out.println(count(myGold, 6));

}

}

**Compilează?** Nu.

**Motivul:** Metoda **count** nu este statică, dar este apelată într-un context static (din **main**). De asemenea, **myGold** este de tip **Short**, iar metoda **count** se așteaptă la doi intregi (**int**).

**Rezolvare:**

1. Schimbăm apelul metodei **count** astfel încât să fie făcut pe o instanță a clasei **Hobbit**, sau facem metoda **count** statică.
2. Convertim **myGold** la **int** când apelăm metoda **count**.

Exemplu de cod corectat:

class Hobbit {

static int count(int x, int y) {

return x + y;

}

}

public class Frodo extends Hobbit {

public static void main(String[] args) {

Short myGold = 7;

System.out.println(count(myGold.intValue(), 6));

}

}

**Subpunctul b)**

public static void main(String[] args) {

String s = "-";

try {

throw new Exception();

}

catch (Exception e) {

try {

try {

throw new Exception();

}

catch (Exception ex) {

s += "ic ";

}

int x = 2;

System.out.println(x / 0);

}

catch (Exception x) {

s += "mc ";

}

finally {

s += "mf ";

}

}

finally {

s += "of ";

}

System.out.println(s);

}

**Motivul:** Există erori de sintaxă. Blocurile **try** și **catch** nu sunt închise corespunzător, există paranteze suplimentare închise și lipsesc altele.

**Rezolvare:**

1. Corectăm parantezele și structura blocurilor **try-catch-finally**.
2. În ceea ce privește diviziunea cu zero, aceasta va arunca o excepție **ArithmeticException**, deci blocul **catch (Exception x)** va fi executat.

Cod corectat:

javaCopy code

public static void main(String[] args) {

String s = "-";

try {

throw new Exception();

}

catch (Exception e) {

try {

try {

throw new Exception();

}

catch (Exception ex) { s += "ic ";

}

int x = 2; System.out.println(x / 0);

}

catch (Exception ex) { s += "mc

}

finally { s += "mf ";

}

}

finally { s += "of ";

}

System.out.println(s);

}

**Ce afișează?**

* Programul va afișa **"ic mc mf of "** deoarece se aruncă mai întâi o excepție, apoi se intră în blocul **catch** interior unde se adaugă **"ic "**. Apoi, diviziunea cu zero aruncă o altă excepție, deci se adaugă **"mc "**. Blocurile **finally** sunt executate întotdeauna, deci se adaugă **"mf "** și **"of "**.

**Dacă x este declarat double?**

* Dacă **x** este un **double**, diviziunea **x / 0** nu va arunca o excepție **ArithmeticException**, ci va rezulta în **Infinity**. Prin urmare, secvența de caractere **"mc "** nu va fi adăugată la **s**, iar output-ul va fi diferit.