Laborator 4 – Limbaje de programare

Reutilizarea codului

1. Sintaxa compoziției

Se plasează o referință către un obiect existent într-o clasă nouă

//: reusing/SprinklerSystem.java

// Composition for code reuse.

class WaterSource {

private String s;

WaterSource() {

System.out.println("WaterSource()");

s = "Constructed";

}

public String toString() { return s; }

}

public class SprinklerSystem {

private String valve1, valve2, valve3, valve4;

private WaterSource source = new WaterSource(); // se face compozitie

private int i;

private float f;

public String toString() { return

"valve1 = " + valve1 + " " +

"valve2 = " + valve2 + " " +

"valve3 = " + valve3 + " " +

"valve4 = " + valve4 + "\n" +

"i = " + i + " " + "f = " + f + " " +

"source = " + source;

}

public static void main(String[] args) {

SprinklerSystem sprinklers = new SprinklerSystem();

System.out.println(sprinklers);

}

} /\* Output:

WaterSource()

valve1 = null valve2 = null valve3 = null valve4 = null

i = 0 f = 0.0 source = Constructed

\*///:

Atentie la metoda toString!!!

2. sintaxa mostenirii – cuvântul cheie extends

//: reusing/Detergent.java

// Inheritance syntax & properties.

import static net.mindview.util.Print.\*;

class Cleanser {

private String s = "Cleanser";

public void append(String a) { s += a; }

public void dilute() { append(" dilute()"); }

public void apply() { append(" apply()"); }

public void scrub() { append(" scrub()"); }

public String toString() { return s; }

public static void main(String[] args) {

Cleanser x = new Cleanser();

x.dilute(); x.apply(); x.scrub();

print(x);

}

}

public class Detergent extends Cleanser {

// Change a method:

public void scrub() {

append(" Detergent.scrub()");

super.scrub(); // Call base-class version

}

// Add methods to the interface:

public void foam() { append(" foam()"); }

// Test the new class:

public static void main(String[] args) {

Detergent x = new Detergent();

x.dilute();

x.apply();

x.scrub();

x.foam();

print(x);

print("Testing base class:");

Cleanser.main(args);

}

} /\* Output:

Cleanser dilute() apply() Detergent.scrub() scrub() foam()

Testing base class:

Cleanser dilute() apply() scrub()

\*///:~

3. Apel constructor clasă de bază cu argumente

//: reusing/Chess.java

// Inheritance, constructors and arguments.

import static net.mindview.util.Print.\*;

class Game {

Game(int i) {

print("Game constructor"); }

}

class BoardGame extends Game {

BoardGame(int i) {

super(i);

print("BoardGame constructor");

}

}

public class Chess extends BoardGame {

Chess() {

super(11);

print("Chess constructor");

}

public static void main(String[] args) {

Chess x = new Chess();

}

} /\* Output:

Game constructor

BoardGame constructor

Chess constructor

\*///:~

4. Creati o clasă de bază care să aibă un constructor cu argumente. Clasa nu are constructor default. Creati o clasă derivată care are atat constructor default (fără argumente) cât și constructor cu argumente. In constructorii clasei derivate, apelati constructorul clasei de bază.

5. se crează o clasă Root care contine instante ale claselor Component1, Component2 si Component3. Se crează o clasă Stem derivată din Root, care la rândul ei agregă din clasele Component1, Component2 si Component3. Fiecare clasă are un constructor default care afisează un mesaj. Arătati ordinea de initializare a obiectelor in cazul combinarii mostenirii cu agregarea.

6. Operatia de delegare

public class SpaceShipControls {

void up(int velocity) {}

void down(int velocity) {}

void left(int velocity) {}

void right(int velocity) {}

void forward(int velocity) {}

void back(int velocity) {}

void turboBoost() {}

} ///:~

public class SpaceShipDelegation {

private String name;

private SpaceShipControls controls =

new SpaceShipControls();

public SpaceShipDelegation(String name) {

this.name = name;

}

// Delegated methods:

public void back(int velocity) {

controls.back(velocity);

}

public void down(int velocity) {

controls.down(velocity);

}

public void forward(int velocity) {

controls.forward(velocity);

}

public void left(int velocity) {

controls.left(velocity);

}

public void right(int velocity) {

controls.right(velocity);

}

public void turboBoost() {

controls.turboBoost();

}

public void up(int velocity) {

controls.up(velocity);

}

public static void main(String[] args) {

SpaceShipDelegation protector =

new SpaceShipDelegation("NSEA Protector");

protector.forward(100);

}

} ///:~

7. Combinarea compozitiei cu mostenirea

//: reusing/PlaceSetting.java

// Combining composition & inheritance.

import static net.mindview.util.Print.\*;

class Plate {

Plate(int i) {

print("Plate constructor");

}

}

class DinnerPlate extends Plate {

DinnerPlate(int i) {

super(i);

print("DinnerPlate constructor");

}

}

class Utensil {

Utensil(int i) {

print("Utensil constructor");

}

}

class Spoon extends Utensil {

Spoon(int i) {

super(i);

print("Spoon constructor");

}

}

class Fork extends Utensil {

Fork(int i) {

super(i);

print("Fork constructor");

}

}

class Knife extends Utensil {

Knife(int i) {

super(i);

print("Knife constructor");

}

}

// A cultural way of doing something:

class Custom {

Custom(int i) {

print("Custom constructor");

}

}

public class PlaceSetting extends Custom {

private Spoon sp;

private Fork frk;

private Knife kn;

private DinnerPlate pl;

public PlaceSetting(int i) {

super(i + 1);

sp = new Spoon(i + 2);

frk = new Fork(i + 3);

kn = new Knife(i + 4);

pl = new DinnerPlate(i + 5);

print("PlaceSetting constructor");

}

public static void main(String[] args) {

PlaceSetting x = new PlaceSetting(9);

}

}

8. Stergerea obiectelor asigurată in clauza finally a try-catch global

//: reusing/CADSystem.java

// Ensuring proper cleanup.

package reusing;

import static net.mindview.util.Print.\*;

class Shape {

Shape(int i) { print("Shape constructor"); }

void dispose() { print("Shape dispose"); }

}

class Circle extends Shape {

Circle(int i) {

super(i);

print("Drawing Circle");

}

void dispose() {

print("Erasing Circle");

super.dispose();

}

}

class Triangle extends Shape {

Triangle(int i) {

super(i);

print("Drawing Triangle");

}

void dispose() {

print("Erasing Triangle");

super.dispose();

}

}

class Line extends Shape {

private int start, end;

Line(int start, int end) {

super(start);

this.start = start;

this.end = end;

print("Drawing Line: " + start + ", " + end);

}

void dispose() {

print("Erasing Line: " + start + ", " + end);

super.dispose(); // super.dispose e ultima instructiune din dispose-ul clasei derivate/compuse

}

}

public class CADSystem extends Shape {

private Circle c;

private Triangle t;

private Line[] lines = new Line[3];

public CADSystem(int i) {

super(i + 1);

for(int j = 0; j < lines.length; j++)

lines[j] = new Line(j, j\*j);

c = new Circle(1);

t = new Triangle(1);

print("Combined constructor");

}

public void dispose() {

print("CADSystem.dispose()");

// The order of cleanup is the reverse

// of the order of initialization:

t.dispose();

c.dispose();

for(int i = lines.length - 1; i >= 0; i--)

lines[i].dispose();

super.dispose();

}

public static void main(String[] args) {

CADSystem x = new CADSystem(47);

try {

// Code and exception handling...

} finally { // aici e finally de la try-catch global

x.dispose();

}

}

}

9. upcast

//: reusing/Wind.java

// Inheritance & upcasting.

class Instrument {

public void play() {}

static void tune(Instrument i) {

// ...

i.play();

}

}

// Wind objects are instruments

// because they have the same interface:

public class Wind extends Instrument {

public static void main(String[] args) {

Wind flute = new Wind();

Instrument.tune(flute); // Upcasting !!! aici e upcast

}

} ///:~