Laborator 5

1. Upcast – se va remarca legarea dinamica a numelor de functii la referintele concrete catre obiecte – rog atentie la numele de fisiere in care trebuie scrise fiecare clasa

public enum Note {

MIDDLE\_C, C\_SHARP, B\_FLAT; // Etc.

} ///:~

class Instrument {

public void play(Note n) {

print("Instrument.play()");

}

}

///:~

// Wind objects are instruments because they have the same interface:

public class Wind extends Instrument {

// Redefine interface method:

public void play(Note n) { // metoda play este cea suprascrisa

System.out.println("Wind.play() " + n);

}

} ///:~

public class Music {

public static void tune(Instrument i) {

// ...

i.play(Note.MIDDLE\_C);

}

public static void main(String[] args) {

Wind flute = new Wind();

tune(flute); // Upcasting

}

}

1. Campurile si metodele statice sunt legate la compilare. Polimorfismul nu functioneaza in cazul campurilor

class Super {

public int field = 0;

public int getField() { return field; }

}

class Sub extends Super {

public int field = 1;

public int getField() { return field; }

public int getSuperField() { return super.field; }

}

public class FieldAccess {

public static void main(String[] args) {

Super sup = new Sub(); // Upcast

System.out.println("sup.field = " + sup.field + ", sup.getField() = " + sup.getField());

Sub sub = new Sub();

System.out.println("sub.field = " + sub.field + ", sub.getField() = " + sub.getField() +

", sub.getSuperField() = " + sub.getSuperField());

}

}

1. In constructor nu se realizează legare dinamică

class Glyph {

void draw() { print("Glyph.draw()"); }

Glyph() {

print("Glyph() before draw()");

draw(); // atentie: care draw va fi apelat?

print("Glyph() after draw()");

}

}

class RoundGlyph extends Glyph {

private int radius = 1;

RoundGlyph(int r) {

radius = r;

print("RoundGlyph.RoundGlyph(), radius = " + radius);

}

void draw() {

print("RoundGlyph.draw(), radius = " + radius);

}

}

public class PolyConstructors {

public static void main(String[] args) {

new RoundGlyph(5);

}

}

1. State design pattern: folositi mostenirea pentru a exprima si coda diferente de comportament si compozitia pentru a exprima variatii de stare

// Dynamically changing the behavior of an object via composition (the "State" design pattern).

class Actor {

public void act() {}

}

class HappyActor extends Actor {

public void act() { print("HappyActor"); }

}

class SadActor extends Actor {

public void act() { print("SadActor"); }

}

class Stage {

private Actor actor = new HappyActor();

public void change() { actor = new SadActor(); }

public void performPlay() { actor.act(); }

}

public class Transmogrify {

public static void main(String[] args) {

Stage stage = new Stage();

stage.performPlay();

stage.change();

stage.performPlay();

}

}

1. Downcast si RTTI – la conversie inversă upcast, in caz de esec se arunca exceptie de tipul ClassCastException

class Useful {

public void f() {}

public void g() {}

}

class MoreUseful extends Useful {

public void f() {}

public void g() {}

public void u() {}

public void v() {}

public void w() {}

}

public class RTTI {

public static void main(String[] args) {

Useful[] x = {

new Useful(),

new MoreUseful()

};

x[0].f();

x[1].g();

// Compile time: method not found in Useful:

//! x[1].u();

((MoreUseful)x[1]).u(); // Downcast/RTTI

((MoreUseful)x[0]).u(); // Exception thrown

}

}