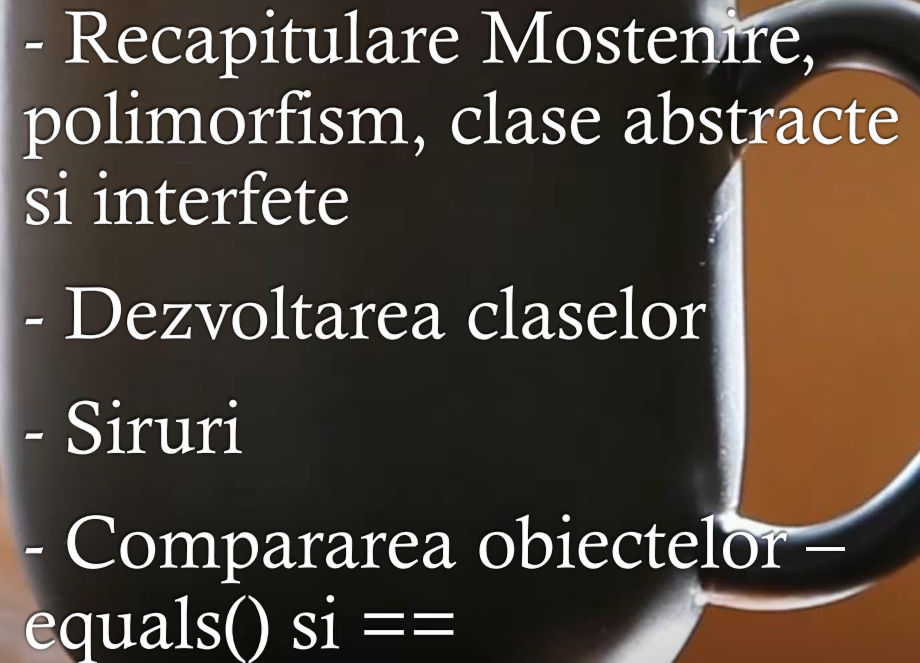


# SDA – Java

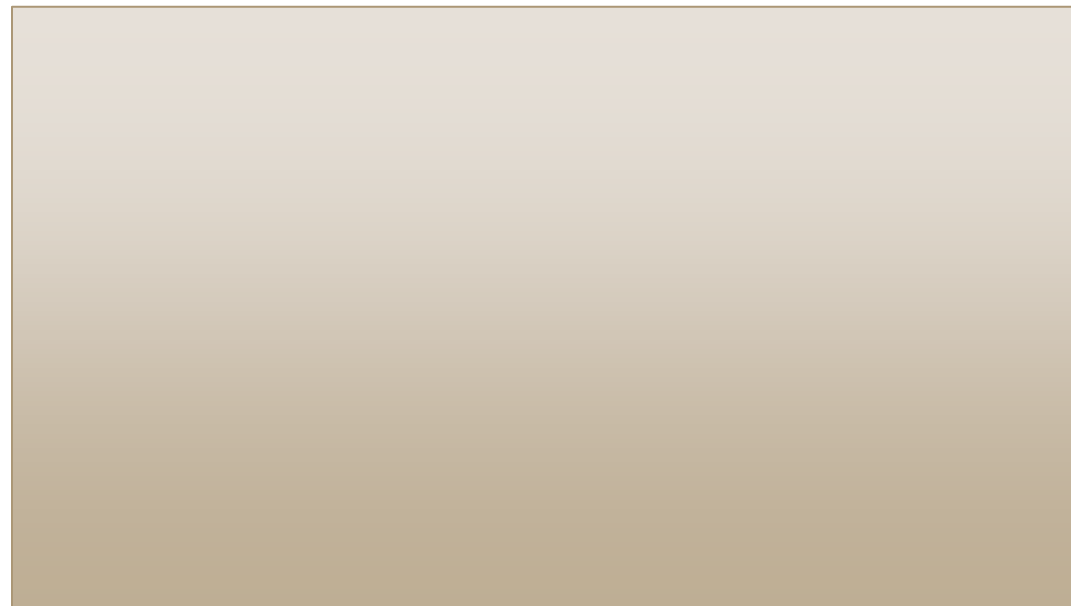
(3)

- 
- Recapitulare Mostenire, polimorfism, clase abstracte si interfete
  - Dezvoltarea claselor
  - Siruri
  - Compararea obiectelor – equals() si ==

```
class Creion {
    public Creion() {
        System.out.println("Creion()");
        scrie();
    }
    public void scrie(){
        System.out.println("HB");
    }
}
class CreionColorat extends Creion {
    private String culoare;
    public CreionColorat(String culoare) {
        System.out.println("CreionColorat()");
        this.culoare = culoare;
        scrie();
    }
    @Override
    public void scrie() {
        System.out.println(culoare);
    }
}
public class Main {
    public static void main(String[] args) {
        CreionColorat crosu = new CreionColorat("rosie");
    }
}
```

# Recapitulare moștenire

Ce se afișează?



```

class Creion {
    public Creion() {
        System.out.println("Creion()");
        scrie();
    }
    public void scrie(){
        System.out.println("HB");
    }
}

class CreionColorat extends Creion {
    private String culoare;
    public CreionColorat(String culoare) {
        System.out.println("CreionColorat()");
        this.culoare = culoare;
        scrie();
    }
    @Override
    public void scrie() {
        System.out.println(culoare);
    }
}

public class Main {
    public static void main(String[] args) {
        CreionColorat crosu = new CreionColorat("rosie");
    }
}

```

# Recapitulare moștenire

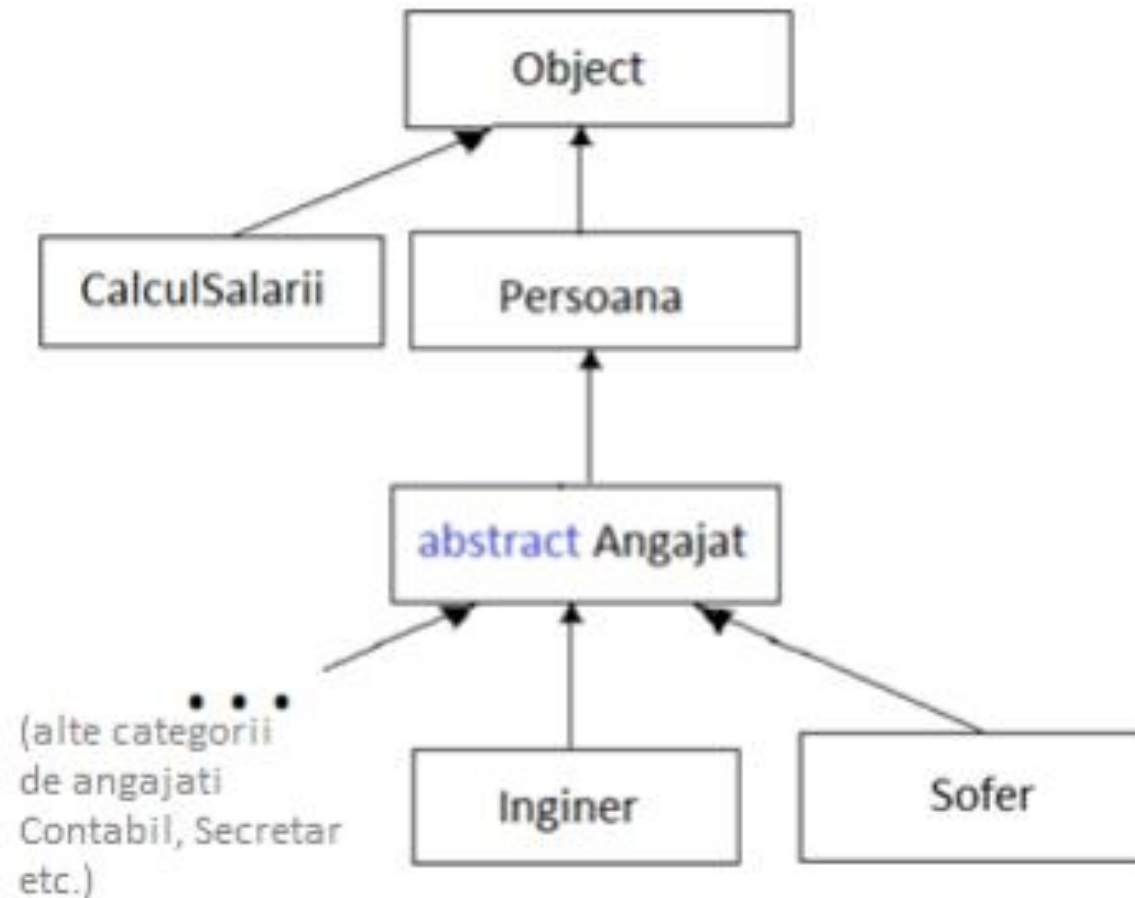
Ce se afisează?

```

Creion()
null
CreionColorat()
rosie

```

Recapitulare  
Mostenire,  
polimorfism,  
clase abstracte si  
interfete



# Recapitulare Mostenire, polimorfism, clase abstracte si interfete

## Interfata IAngajat

Această interfață va fi implementată în mod obligatoriu de clasele **Inginer** și **Șofer**.

```
public interface IAngajat {  
    public static final double salariuOrarMinim=15;  
    public void setNrOreLucrate(int nrOreLucrate);  
    public double salariu();  
}
```

**Nota.** In interfetele Java variabilele sunt **implicit** `public static final` deoarece

- `public` - trebuie sa fie vizibile in toate clasele care implementeaza interfata
- `static` - trebuie sa existe inainte de orice instanta a unei clase care o implementeaza
- `final` - fiind comune tuturor claselor care implementeaza interfata ele nu trebuie sa poata fi modificate

De aceea prezenta in declaratie a `public static final` este redundanta (declaratia putea fi doar `double salariuOrarMinim=15;` )

```
public class CalculSalarii {  
    public static void main(String arg[]){  
        Inginer ing1=new Inginer( nume: "Barbulescu", prenume: "Barbu", id: 101);  
        Inginer ing2=new Inginer( nume: "Trestie", prenume: "Tudor", id: 23);  
        Sofer sofr= new Sofer( nume: "Repede", prenume: "Raul", id: 302);  
  
        IAngajat[] salariati = new Angajat[]{ing1,ing2,sofr};  
    }  
}
```



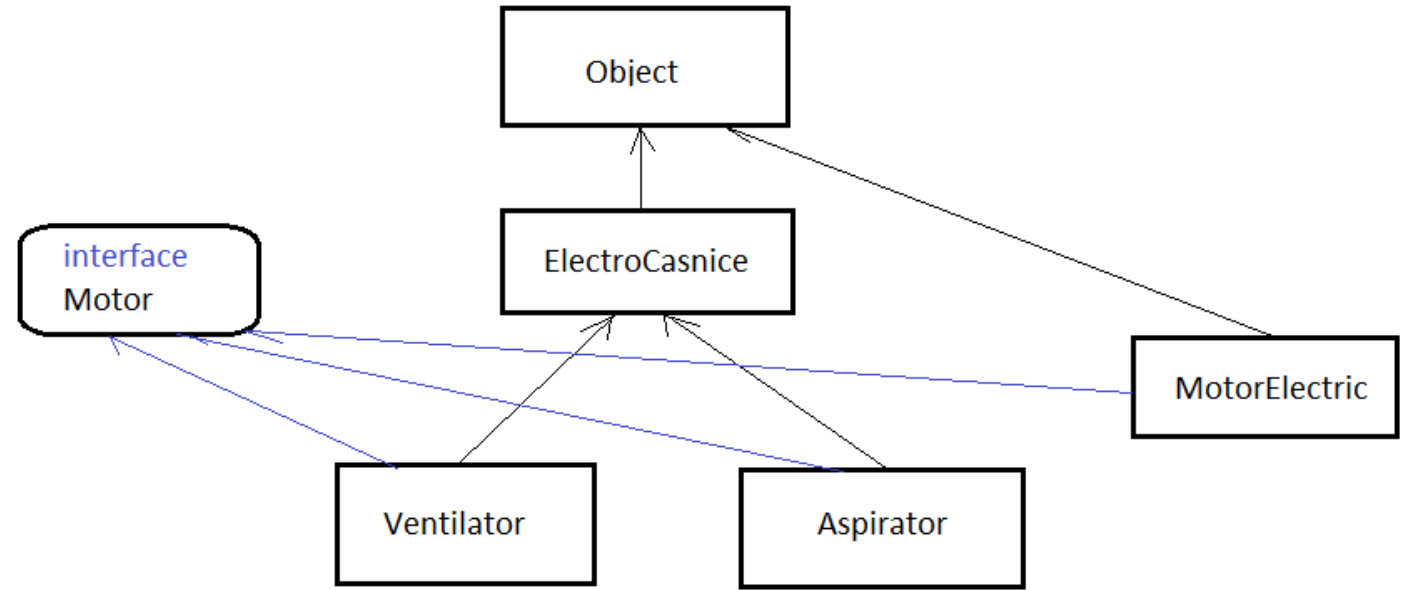
## Exemplu de utilizare a interfetelor

Clasele Ventilator și Aspirator

- extind Electrocasnice și
- implementează interfața Motor.

Se pot construi 2 tabele polimorfe

*Alături este conținutul funcției main() din clasa principală și rezultatul afișat.*



```
MotorElectric jucarie = new MotorElectric();
Aspirator a = new Aspirator(500);
Ventilator v = new Ventilator(200);
```

```
ElectroCasnice[] aparate = new ElectroCasnice[] { a, v};
double costTotal=0;
for ( ElectroCasnice apcrt : aparate )
    costTotal += apcrt.getPret();
System.out.println(costTotal);
```

```
Motor[] motoare = new Motor[] {jucarie, a, v};
for ( Motor mcrt : motoare )
    mcrt.start();
```

700.0  
ME Start  
Aspirator pornit  
Ventilator pornit

```
package ro.usv;
```

```
public interface Motor {  
    void start();  
    void stop();  
}
```

```
package ro.usv;
```

```
public class MotorElectric implements  
Motor {  
    @Override  
    public void start() {  
        System.out.println("ME Start");  
    }  
  
    @Override  
    public void stop() {  
        System.out.println("ME Stop");  
    }  
}
```

```
package ro.usv;
```

```
public class ElectroCasnice {  
    private double pret;
```

```
    public ElectroCasnice(double pret) {  
        this.pret = pret;  
    }
```

```
    public double getPret() {  
        return pret;  
    }
```

```
    public void setPret(double pret) {  
        this.pret = pret;  
    }
```

```
    @Override  
    public String toString() {  
        return getClass().getSimpleName() + " pret=" + pret;  
    }  
}
```

```
package ro.usv;
public class Aspirator extends ElectroCasnice
    implements Motor {
    //ce se intampla daca nu ar fi implements Motor?

    public Aspirator(double pret) {
        super(pret);
    }

    @Override
    public void start() {
        System.out.println("Aspirator pornit");
    }

    @Override
    public void stop() {
        System.out.println("Aspirator oprit");
    }
}
```

```
package ro.usv;

public class Ventilator extends ElectroCasnice
    implements Motor{

    public Ventilator(double pret) {
        super(pret);
    }

    @Override
    public void start() {
        System.out.println("Ventilator pornit");
    }

    @Override
    public void stop() {
        System.out.println("Ventilator oprit");
    }
}
```



# Dezvoltarea claselor

- ◆ Moștenire (class **Pixel** extends **Punct** { .....} )
- ◆ Compunere class **Cerc** {  
    Punct centru;  
    double raza;  
    //...  
}
- ◆ Delegare class **MotorElectric** { .... }  
class **Aspirator** extends **ElectroCasnice** {  
    **MotorElectric** motor;  
    public void start(){  
        motor.start();  
    }  
    // ...  
}

# Compararea obiectelor

`obj1 == obj2`

se verifică egalitatea referințelor

`obj1.equals(obj2)`

se ia în considerare egalitatea  
conținutului obiectelor

```
Integer obj1 = new Integer(10);  
Integer obj2 = new Integer(10);  
System.out.println("obj1==obj2 ->" + (obj1==obj2));  
System.out.println("obj1.equals(obj2) ->" + obj1.equals(obj2));
```

`obj1==obj2 ->false`

`obj1.equals(obj2) ->true`

# Sirurile sunt imutabile

În exemplul alăturat s0 este o referință la String inițializată cu aceeași adresă cât este și în s (“2000”).

În urma execuției instrucțiunii

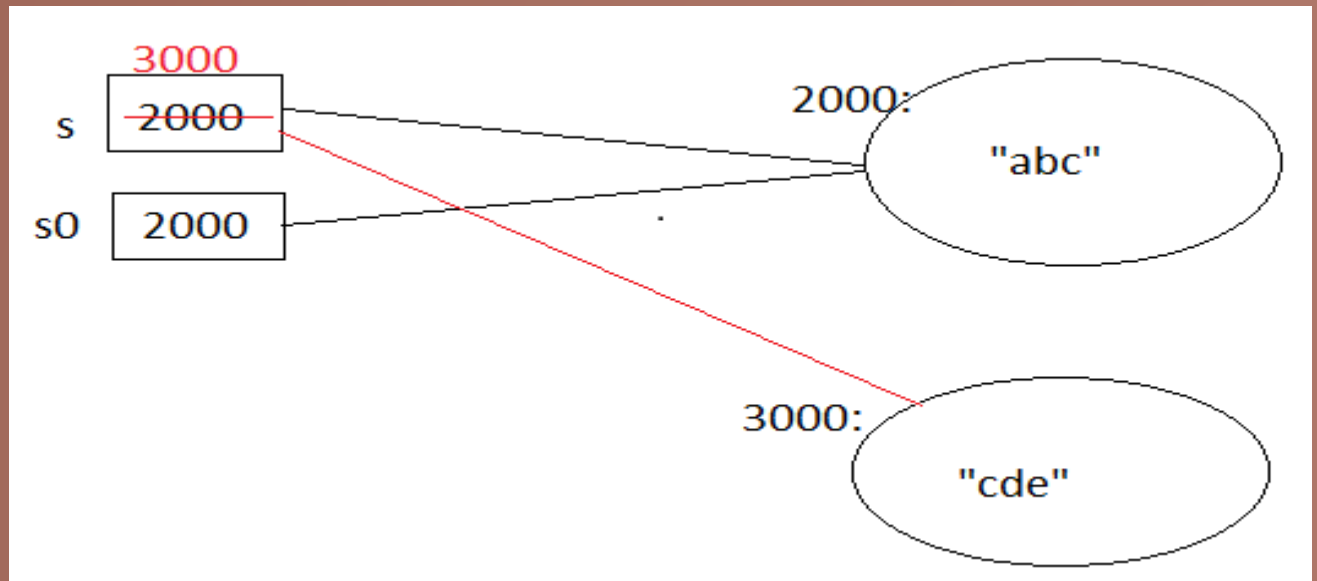
`s = "cde";`

se creează un nou obiect de tip String (la adresa “3000”) deoarece obiectul de la adresa “2000” nu poate fi modificat.

În s0 s-a păstrat însă adresa primului obiect de tip String

```
String s = "abc";    // String s = new String("abc");  
String s0=s;  
s = "cde";  
System.out.println("s="+s+" s0="+s0 );
```

**s=cde s0=abc**



# Compararea a două șiruri

- boolean equals()
- boolean equalsIgnoreCase()
- int compareTo( *altSir* ) similar cu strcmp() din biblioteca C.
- int compareToIgnoreCase(*altSir*)

## Observatii

- 1) Reamintire: operatorul == verifică egalitatea referințelor nu și a conținutului.
- 2) Rezultatul lui compareTo():  
"abc" > "Abc" deoarece codul lui 'a' (0x61) este mai mare decat codul lui 'A' (0x41)

```
String s0 = "abc";
String sx="Abc";
System.out.println("s0=" + s0 + " sx="+sx);
System.out.println("s0==sx ->" + (s0==sx) );
System.out.println("s0.equalsIgnoreCase(sx) ->" +
                    s0.equalsIgnoreCase(sx) );
System.out.println("s0.equals(sx) ->" + s0.equals(sx) );
System.out.println("s0.compareTo(sx) ->" + s0.compareTo(sx) );
System.out.println("s0.equals(sx.toLowerCase()) ->" +
                    s0.equals ( sx.toLowerCase() ) );
System.out.println("sx=" + sx);
//toLowerCase() nu modifica sirul
```

```
s0=abc    sx=Abc
s0==sx    ->false
s0.equalsIgnoreCase(sx) ->true
s0.equals(sx) ->false
s0.compareTo(sx) ->32
s0.equals(sx.toLowerCase()) ->true
sx=Abc
```



# De ce sunt imutabile şirurile ?

## 1. Optimizare management memorie

La crearea unui nou obiect String, JVM caută dacă nu cumva există deja alocat un şir cu aceeaşi. Dacă există nu se mai alocă un alt obiect, ci se returnează adresa (referinţă) obiectului existent (dovadă în marcajul galben).

Dacă şirurile nu ar fi fost imutabile nu se putea face aşa ceva deoarece în timpul execuţiei s-ar fi putut modifica şirul.

## 2. Securitate

Se evită modificarea accidentală din alt fir de execuţie (Thread) sau chiar din altă funcţie unde şirul este argument.

```
String s1 = "FIESC";
String s2 = "FIESC"; // nu se creează un obiect nou, căci
                    // există deja un şir "FIESC"; de aceea s2 va avea
                    // aceeaşi valoare ca s1 (referinţe egale)
                    // Acesta este şi unul din motivele pt. care
                    // şirurile sunt imutabile.
                    // JVM este apreciată ca o "masterpiece" !

String s3="USV";
System.out.println("s1="+s1+ " , s2="+s2+ " , s3="+s3);
System.out.println("s1==s2 ->"+(s1==s2) + " , s1.equals(s2) ->"+
                    s1.equals(s2));
System.out.println("s1==s3 ->"+(s1==s3) + " , s1.equals(s3) ->"+
                    s1.equals(s3)); // firesc
```

```
s1=FIESC, s2=FIESC, s3=USV
s1==s2 ->true, s1.equals(s2) ->true
s1==s3 ->false, s1.equals(s3) ->false
```



- Siruri

- Compararea  
obiectelor

- equals() si ==

◆ <http://apollo.eed.usv.ro/~pentiuc/sd/comparareObiecte.pdf>