C13: Java

Cuprins:

- Derivare, constructori
- Supradefinirea si supraincarcarea metodelor
- Incapsularea datelor; modificatori de acces
- Clase abstracte si interfete
- Applet – nu intra in materia pentru examen

C13: Java

Derivarea/Mostenirea

- in Java nu este permisa mostenirea multipla (vom vedea cum este compensata aceasta constrangere)
- pentru a deriva dintr-o clasa se foloseste cuvantul cheie **extends** (este extins un tip de date de baza)

```
class Baza {
  //atribute si metode
}

class Derivata extends Baza {
  //atribute si metode mostenite
  //atribute si metode in plus
}
```

- Relatia intre clase este de tip “is a” : Derivata “is a” Baza

C13: Java

- clasele in Java sunt derivate din clasa Object => automat orice clasa pe care o definim are o superclasa (Baza e derivata din Object; Derivata este derivata din Baza)

- in cazul in care nu specificam printr-o clauza extends faptul ca o clasa ar fi derivata din alta, ea e automat derivata din Object

Object este o clasa speciala, deoarece:

- este singura clasa care nu are superclasa;

- metodele definite in Object pot fi supraincarcate/apelate in orice clasa /de catre orice obiect Java.

Deoarece toate clasele au o superclasa si nu este permisa mostenirea multipla, clasele Java formeaza o ierarhie de clase ce poate fi reprezenta ca ***un arbore in care Object este radacina***.

In C++, unde se permite mostenirea multipla, ierarhia este de tip graf aciclic.

Clase derivate – constructori

Considerati urmatoarea clasa de baza:

```
class Baza {
protected int atr1;

Baza(int i){   //constructor cu parametru
     atr1=i;
}

Baza(){    //constructor fara parametri
     this(1);    //apel constructor cu parametru
}

void afisare(){
    System.out.println("atr1 "+atr1);
}

}
```

Observatie: In Java nu se genereaza automat constructor de copiere.

Vrem sa derivam din aceasta clasa:

```
class Derivata extends Baza{//mosteneste atributul atr1 si metoda afisare din Baza
private int atr2;
/*
Derivata(int i, int j)
{//automat, primul lucru care se intampla la intrarea in constructorul clasei derivate este
//apelul automat al constructorului fara parametri din clasa de baza
//daca acesta nu era implementat am fi avut o eroare
    atr1=i; //super.atr1=i;  //super - referire la clasa de baza
    atr2=j;
}*/

//DAR dorim sa reutilizam codul deja implementat in clasa de baza pentru a initializa
//atributele mostenite din aceasta:

Derivata(int i,int j)
{//pe prima linie a constructorului din derivata trebuie apelat constructorul dorit din clasa
//de baza;
    super(i);
    atr2=j;
}

Derivata(){
   this(1,0);
}
}//ce functii mai are clasa Derivata?
```

Cuvantul super

• Cuvantul **super** este rezervat in Java

• El semnifica **referinta la superclasa** (este una singura) si **poate fi utilizat la apelul unui constructor al superclasei** (asa cum this era utilizat si pentru a apela un constructor al clasei in alt constructor al aceleiasi clase).

•Restrictiile asupra utilizarii lui **super** pentru apelul unui constructor sunt:

- se poate utiliza doar intr-o metoda constructor;

- apelul constructorului superclasei trebuie sa apara **ca prima instructiune** din metoda constructor, chiar inainte de declararea unor posibile variabile sau alte operatii.

Constructorul implicit

• Daca o clasa derivata nu defineste niciun constructor, se generaza automat un constructor pentru acea clasa.

• Acest constructor implicit cheama constructorul default al superclasei (daca acesta nu exista, avem o eroare).

Inlantuirea apelurilor constructorilor

• Java garanteaza ca la crearea unei instante a clasei (unui obiect) se apeleaza constructorul clasei.

• Se mai garanteaza ca se apeleaza constructorul superclasei la orice creare a unei instante a subclasei.

• Pentru aceasta, mediul Java trebuie sa forteze ca orice metoda constructor sa apeleze metoda constructor a superclasei:

Daca prima instructiune din constructor nu este un apel explicit al constructorului superclasei, compilatorul Java insereaza implicit apelul super(), deci apeleaza un constructor fara parametri al superclasei.

- Exista o exceptie de la invocarea implicita **super()** si anume: daca prima instructiune a unui constructor este un apel catre alt constructor al aceleiasi clase prin sintaxa this().

- Chiar daca metodele constructor din aceeasi clasa se pot apela unele pe altele, pana la urma una din ele trebuie sa invoce (explicit sau implicit) constructorul superclasei.

Metode supradefinite (overload)

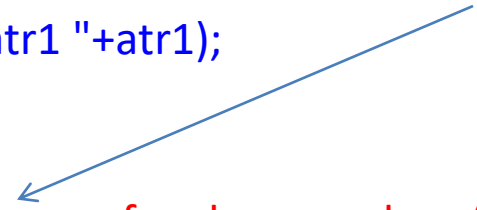
- metode cu acelasi nume, dar cu semnaturi diferite

Metode supraincarcate/suprascrise (override)

- metode cu acelasi nume, aceeasi semnatura si acelasi tip de date returnat (in superclasa si subclasa)

```
class Baza {
protected int atr1;

Baza(){
this(0);}

Baza(int i){//constructorii sunt supradefiniti
atr1=i;
}

void afisare(){
System.out.println("atr1 "+atr1);
}
}
```

```
class Derivata extends Baza{
private int atr2;

Derivata(int i,int j){
 super(i); // reutilizam codul din Baza
 atr2=j;
}

void afisare(){//supraincarca afisare din Baza
 //vrem sa reutilizam codul din Baza
 super.afisare();
 System.out.println("atr2 "+atr2);
}
```



In acest context **super** se refera la superclasa (mereu una singura) si apeleaza metoda de afisare din Baza – reutilizarea codului; altfel nu puteam sa avem acces la ea in clasa derivata deoarece are acelasi nume ca metoda care o supraincarca. Putem sa o apelam si altfel?

```
public static void main(String []args){ //main-ul se gaseste in clasa Derivata

Baza b=new Baza(10); //declar un obiect de tip Baza si il instantiez
b.afisare();     // atr1 10

Derivata d=new Derivata(1,1);
d.afisare();    //  atr1 1
                //   atr2 1

//obiectele derivate sunt de tip Baza, deci sunt permise urmatoarele:
Baza b1=new Derivata(2,2); //declar un handler de tip Baza si il instantiez cu un obiect Derivata
b1.afisare();  //  atr1 2
               //   atr2 2

Baza b2=d;  //atentie – atribuire intre handleri
b2.afisare(); //  atr1 1
              //   atr2 1

//Ce puteti observa?

//Derivata d1=b;    //o astfel de atribuire nu e permisa
}
}
```

- Observati analogia cu metodele virtuale din C++.
- In Java, diferenta este ca acesta este mecanismul standard de supraincarcare.
- Folosind terminologia de la C++, putem spune ca in Java toate metodele pot avea comportament virtual (cu exceptia metodelor declarate static sau private).

```
Baza []v=new Baza[3]; //vector cu 3 handleri de tip Baza
v[0]=new Baza(2);
v[1]=new Derivata(2,2);
v[2] =new Baza(3);
for (int i=0;i<3;i++)
        v[i].afisare(); // dynamic binding
```

- Daca avem un tablou de obiecte de tip Baza si Derivata, cum stie compilatorul sa cheme functia de afisare() specifica pentru un obiect dat din tablou?
- Compilatorul nu are cum sa stie care este tipul obiectului cu handlerul v[i].
- Codul e generat si se amana luarea acestei decizii pana in momentul executiei, ceea ce se numeste cautarea dinamica a metodelor (dynamic method lookup).
- Interpretorul Java stie care elemente din tablou sunt obiecte de tip Baza si care de tip Derivata, deci este capabil sa apeleze metoda afisare() adecvata in functie de tipul fiecarui obiect cu handler v[i].

- Aceasta tehnica se numeste "late/dynamic binding" (legare tarzie/dinamica).

- Cautarea dinamica este rapida, dar nu atat de rapida pe cat este apelul direct al unei metode - legare statica.

Exista unele situatii in care legarea dinamica nu e necesara:

- **Metodele statice nu pot fi supraincarcate**, deci sunt intotdeauna *invocate direct*.

- **Metodele private nu sunt mostenite in subclasa** si deci nu pot fi supraincarcate in aceasta=> vor fi *invocate direct*.

- Metodele sunt *invocate direct* daca **apelul lor se face de catre un obiect cu handler de acelasi tip** (ex: Derivata x=new Derivata(); x.afisare();).

- In cazul in care **nu vreau sa supraincarc o metoda** a unei clase, as vrea sa fie **folosita legarea statica**, as avea nevoie de un mecanism ca sa precizez faptul ca nu va putea fi suprascrisa -> **final**

Atributul final

Daca o clasa este declarata ***final***, ea nu mai poate fi extinsa prin derivare.

```
final class Baza
{//…
}

class Derivata extends Baza
{
}//error
```

Daca o metoda este declarata ***final***, ea nu mai poate fi suprascrisa intr-o clasa derivata.
Constructorii nu pot sa fie declarati ***final.***
Toate metodele statice , private si metodele unei clase finale <=>legare statica (si sunt finale).

```
class Baza
{ final int f(){
      return 1;
  }
}

class Derivata extends Baza
{ int f(){ //error
     return 2;
  }
}
```

Variabile “ascunse”/”shadowed”

```
class Baza {
protected int atr;

Baza(int i){
  atr=i;
}

void afisare(){
  System.out.println("atr "+atr);
}
}
```

```
class Derivata extends Baza{
//mosteneste atributul atr din Baza
private int atr;
//acopera ca vizibilitate atributul atr din Baza

Derivata(int i,int j)
{ super(i);
 atr=j; //atr declarat in Derivata
}

void afisare()
{//System.out.println("atr baza"+super.atr);
//sau
 System.out.println("atr baza"+((Baza)this).atr);

System.out.println("atr derivata "+atr);
}
//cum mai pot sa implementez functia de afisare?

}
```

Daca dorim sa avem acces la atributul ascuns din Baza trebuie sa facem precizarea clara:
super.atr;

SAU sa convertim obiectul curent la tipul Baza si apoi sa accesam atributul:
((Baza)this).atr;

Observatie: Daca C este subclasa a lui B, care este subclasa a lui A, si C contine un atribut x care apare si in B si in A, atunci ne putem referi din clasa **C** la aceste atribute in modul urmator:

```
x                   // Atributul x din C
this.x              // Atributul x din C
super.x             // Atributul x din B
((B) this).x        // Atributul x din B
((A) this).x        // Atributul x din A
super.super.x       // ILEGAL: NU se refera la x din A
```

NU se poate extinde sintaxa cu super pe mai mult de un nivel.

Cele 4 principii de baza ale programarii orientate pe obiect

* Incapsularea datelor **Abstractizare

***Mostenire ****Polimorfism

Incapsularea datelor

- Un principiu de baza in programarea orientata obiect este **incapsularea** datelor, adica **restrictionarea accesului** la date numai prin intermediul unor metode dedicate.
- Ratiunea acestei tehnici este protectia clasei impotriva unor modificari accidentale (erori).
- O clasa contine adesea un numar de atribute care sunt interdependente si care trebuie sa fie intr-o stare consistenta.
- Daca se permite unui programator sa manipuleze direct aceste atribute, e posibil sa se piarda consistenta. Daca insa trebuie apelata o metoda pentru a modifica un atribut, acea metoda va face toate operatiile necesare pastrarii consistentei.

Astfel trebuie pusa problema controlului vizibilitatii datelor si metodelor.

Metode pentru acces la date

- Este indicat ca atributele clasei sa nu poata fi modificate/accesate direct din afara clasei, ci prin intermediul unor metode de acces.

- Un pachet este un grup de clase inrudite care pot coopera.

- Toate atributele si metodele neprivate din toate clasele pachetului sunt vizibile oriunde in pachet.

- Singura problema apare atunci cand nu se foloseste declaratia **package**.

- Clasele de acest tip sunt grupate impreuna cu alte clase care nu au specificat un pachet, si toate metodele si datele non-private sunt vizibile in acest pachet default.

Modificatori de acces

- **public -** cand o clasa este declarata **public**, inseamna ca acea clasa e vizibila oriunde. De asemenea, atributele si metodele declarate **public** sunt vizibile oriunde.

- **private -** o variabila declarata cu atributul **private** va fi vizibila doar in metodele definite in clasa respectiva. O metoda **private** poate fi apelata doar din metode care apartin aceleiasi clase. Metodele si datele **private** nu sunt accesibile in clase derivate.

- **protected -** atributele si metodele **protected** sunt vizibile in pachetul curent si in toate subclasele clasei in care au fost declarate.

- **nivelul predefinit de vizibilitate** (in situatia in care nu se scrie nici un atribut de vizibilitate) – vizibilitate oriunde in pachet

Cateva reguli intuitive simple pentru atributele de vizibilitate:

Utilizati **private** pentru metodele si datele care sunt folosite doar in cadrul clasei si care trebuie ascunse in afara.

Utilizati **public** pentru metode, constante si alte variabile importante care trebuie sa fie vizibile oriunde.

Utilizati **protected** daca, dimpotriva, doriti ca toate clasele din acelasi pachet sa aiba acces la campurile respective.

Utilizati **protected** daca doriti ca metodele si variabilele sa fie vizibile SI in subclasele din afara pachetului.

Utilizati **vizibilitatea implicita** pentru metodele si variabilele care doriti sa fie ascunse pentru clasele din afara pachetului, dar care sa fie accesibile claselor din acelasi pachet.

Utilizati declaratia **package** pentru a grupa clasele in pachete.

```
package exemplu;
//clasa Persoana apartine pachetului
//exemplu;

public class Persoana {
protected String nume;
protected int varsta;
//vizibile in pachet si in clasele derivate din
//afara pachetului

Persoana(String n, int v)
{nume=new String(n);
 varsta=v;
}

Persoana(Persoana p)//constr. de copiere
{this(p.nume, p.varsta);
 }

void clone(Persoana p)
{nume=new String(p.nume);
 varsta=p.varsta;
 }

   final void setNume(String n)
   {nume=new String(n);
   }

   final void setVarsta(int v)
   {varsta=v;
   }

   final String getNume()
   {return nume;
   }

   final int getVarsta()
   {return varsta;
   }
//declarate final pentru ca nu o sa vreau sa
//le schimb implementarea in clasele derivate
//si vreau la apel legare statica
   void afisare()
   {System.out.print("Nume:" +nume+ "
            Varsta:"+varsta);
    }
//vreau sa o supraincarc in clasele derivate
}
```

```
package exemplu;
//clasa Persoana apartine pachetului
//exemplu; vizibila doar in pachet

 class Student extends Persoana{
 private String facultate;
 //vizibil doar in Student

 Student(String n, int v, String f)
 {super(n,v);
  facultate=new String(f);
 }

 Student(Student s)
 {super(s.nume,s.varsta);
  facultate=new String(s.facultate);
//sau
//this(s.nume,s.varsta,s.facultate);
 }

 void clone(Student s)
 {super.clone(s);
 facultate=new String(s.facultate);
 }

final void setFacultate(String f)
{facultate=new String(f);
}

final String getFacultate()
{return facultate;
}

void afisare()
 {super.afisare();
  System.out.print(" Facultate "+facultate);
 }

//-clasa nu e declara final; se poate deriva in
//continuare clasa Student_Angajat; dar
//atentie la vizibilitatea atributului facultate
//-pot sa fie suprascrise metodele clone si
//afisare
```

Realizati un vector in care sa stocati atat persoane cat si studenti.
Sortati-l dupa nume si afisati toate atributele.

```
public static void main(String []args) //in clasa Student; dar putea fi in orice alta clasa
{int n=10;
 Persoana [] p= new Persoana[n];  //creez n handleri de tip Persoana; nu se apeleaza
                                  // constructorul din Persoana

 for (int i=0;i<n;i++)
            if (i%2==1) p[i]=new Persoana("Cineva"+(n-i),i+18);
            else  p[i]=new Student("Cineva"+(n-i),i+18,"A&C");

 //sortare dupa nume
 for (int i=0;i<n-1;i++)
            for (int j=i+1;j<n;j++)
                        if ((p[i].getNume()).compareTo(p[j].getNume())>0){
                                    Persoana x=p[i]; //atribuiri de handleri
                                    p[i]=p[j];
                                    p[j]=x;
                        }

  for (int i=0;i<n;i++) {
            p[i].afisare();
            System.out.println();
  }
} //main
} //clasa
```

```
Persoana pp=new Student("sa",1,"sa");
System.out.println(pp.getClass());//class exemplu.Student
//getClass() - metoda mostenita din clasa Object
//returneaza tipul obiectului cu handlerul pp


//Student ss=new Persoana(“sa”,10); //atribuirea nu este permisa
// Persoana nu este in Student


Student ss=(Student)pp; //da, deoarece pp se refera la un obiect de tip Student
ss.afisare();
System.out.println( ss.getClass()); //class exemplu.Student
```

Clase abstracte si interfete

Se presupunem ca dorim sa implementam un numar de clase de tip figuri geometrice 2D: Dreptunghi, Patrat, Cerc, Triunghi etc.

Vrem ca toate clasele sa aiba implementate metodele arie() si perimetru().

Pentru a facilita lucrul cu containere ce contin figuri, ar fi util daca toate clasele geometrice ar fi subclase ale unei clase de baza *Shape*.

Vrem ca Shape sa incapsuleze toate caracteristicile si comportamentele comune ale claselor, adica metodele arie() si perimetru().

Dar clasa Shape nu poate implementa efectiv aceste metode, deoarece ea nu reprezinta o figura geometrica concreta.

Java trateaza aceasta situatie (similar cu C++) -> **metode abstracte**.

Metode si clase abstracte

•Java permite definirea unei metode fara implementare prin declararea ei ca metoda **abstracta**.

•O metoda **abstracta** nu are corp, ci numai semnatura, urmata imediat de **;**

•Metodele abstracte Java sunt similare functiilor virtuale pure (abstracte) din C++.

Observatii:

•orice clasa cu o metoda abstracta este automat abstracta; o clasa declarata **abstract** trebuie sa aiba cel putin o metoda abstracta;

•o clasa abstracta nu poate fi instantiata;

•o subclasa a unei clase abstracte poate fi instantiata daca suprascrie (implementeaza) toate metodele abstracte ale superclasei

•daca o subclasa a unei clase abstracte nu implementeaza toate metodele abstracte pe care le mosteneste, acea clasa este ea insasi abstracta.

```
public abstract class Shape {        //clasa abstracta
    protected static final double PI = 3.14;//constanta – folosita in comun de toate obiectele

    public abstract double arie(); //metode abstracte
    public abstract double perimetru();
}

class Cerc extends Shape {
            protected double r;
            public Cerc() { r = 1.0; }
            public Cerc (double r) { this.r = r; }
            public double arie() { return PI * r * r; }//daca nu implementam; Cerc –clasa abstracta
            public double perimetru() { return 2.0 * PI * r; }
            final public double getRaza(){ return r;}
}
class Dreptunghi extends Shape {
            protected double w, h;
            public Dreptunghi() { w = 0.0; h = 0.0; }
            public Dreptunghi(double w, double h) { this.w = w; this.h = h; }
            public double arie() { return w * h; }
            public double perimetru() { return 2.0 * (w + h); }
            final public double getW() { return w; }
            final public double getH() { return h; }
```

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

        Shape [] shapes = new Shape [3];
        shapes[0] = new Cerc(2.0);
        shapes[1] = new Dreptunghi(1.0, 2.0);
        shapes[2] = new Dreptunghi(4.0, 5.0);
        double ariaTotala = 0.0;
        for(int i = 0; i < shapes.length; i++)
                ariaTotala += shapes[i].arie();

        System.out.println(ariaTotala);
        }

}
```

Interfete

- In Java nu avem decat mostenire simpla
- Sa presupunem ca dorim o aplicatie cu figuri geometrice care sa poata fi desenate (au culoare specifica si sunt pozitionate undeva in spatiul 2D si pot sa fie desenate).
- Am putea defini o noua clasa abstracta *DrawableShape* si apoi implementa diverse subclase, cum ar fi CercDesenabil, DreptunghiDesenabil etc., ceea ce ar functiona corect.

- Dar dorim ca formele desenabile sa aiba metodele arie() si perimetru(), pe care nu vrem sa le reimplementam.
- Asta ar insemna ca CercDesenabil sa fi subclasa si a lui Cerc, ceea ce nu e posibil (e deja derivat din Shape).

- Solutia Java este crearea de ***interfete***.
- O interfata seamana mult cu o clasa abstracta, dar foloseste cuvantul cheie ***interface*** in loc de ***abstract class*** si toate metodele sale sunt abstracte.

Iata o interfata numita Drawable:

```
public interface Drawable {
        public void setCuloare(Color c);
        public void setPozitie(double x, double y);
        public void desenare();
}
```

In timp ce o clasa abstracta poate defini atat metode abstracte cat si normale, toate **metodele definite intr-o interfata sunt implicit abstracte**.

Putem folosi cuvantul cheie abstract pentru toate metodele, dar el este implicit.

Orice variabila declarata intr-o interfata trebuie sa fie **statica si finala**.

O interfata nu are constructori.

Cum o clasa poate extinde o clasa abstracta, tot asa ea poate **implementa** o interfata.

Pentru aceasta se foloseste cuvantul-cheie **implements**, urmat de numele interfetei, scris dupa o eventuala clauza extends.

O asemenea clasa ce implementeaza o interfata trebuie sa asigure implementari pentru toate metodele din interfata, altfel e considerata abstracta.

```
class CercDesenabil extends Cerc implements Drawable {
 // atribute private noi
private Color c;
private double x, y;

// Constructor
public CercDesenabil (double r, double x, double y, Color c)
{ super(r);
this.x=x;
this.y=y;
this.c=new Color(c);
}

// Se mostenesc toate metodele din Cerc

// Implementarea interfetei
public void setCuloare(Color c) {
this.c = new Color(c); }

public void setPozitie(double x, double y) {
this.x = x; this.y = y; }

public void desenare(){System.out.println("Culoarea: "+c.toString()+" Pozitie "+ x+" " +y );}
}
```

```
public interface Drawable {
 public void setCuloare(Color c);
 public void setPozitie(double x, double y);
 public void desenare();
}
```

```
class DreptunghiDesenabil extends Dreptunghi implements Drawable {
private Color c;
private double x, y;

public DreptunghiDesenabil (double w, double h,double x, double y, Color c) {
super(w, h);
this.x=x;
this.y=y;
this.c=new Color(c);
}

public void setCuloare(Color c) {
this.c = new Color(c);
 }

public void setPozitie(double x, double y) {
this.x = x;
this.y = y;
}

public void desenare(){
System.out.println("Culoarea: "+c.toString()+" Pozitie "+ x+" " +y );
}
```

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

        Drawable [] vec= new Drawable [3];
        vec[0] = new CercDesenabil(2.0, 30,30,Color.BLACK);
        vec[1] = new DreptunghiDesenabil(1.0, 2.0, 10,10, Color.RED);
        vec[2] = new DreptunghiDesenabil(4.0, 5.0,20,20, Color.BLUE);

        for(int i = 0; i < shapes.length; i++) {
                vec[i].setPozitie(i,i);
                vec[i].desenare();}
}
}
```

- Trebuie neaparat sa accesam un obiect CercDesenabil printr-un handler de tip Drawable pentru a putea apela metoda desenare(), sau printr-un handler Shape pentru a putea apela metoda arie().

Implementarea mai multor interfete

•O clasa poate implementa mai multe interfete.

•De exemplu, sa presupunem ca mai exista o interfata numita Scalable care defineste mecanisme de scalare a figurilor geometrice.

•Sintaxa unei noi clase care extinde Dreptunghi si implementeaza Drawable si Scalable ar fi:

```
public class DreptunghiDesenabilScalabil extends Dreptunghi
        implements Drawable, Scalable {
                //....
}
```

Extinderea interfetelor

•Interfetele pot avea subinterfete, asa cum clasele au subclase.

•O subinterfata mosteneste toate metodele abstracte si constantele din superinterafata sa, si poate defini in plus alte metode si constante.

Totusi, exista o diferenta importanta fata de subclase:

o interfata poate extinde mai multe interfete

```
public interface Transformable extends Scalable, Reflectable { }
```

•Se mostenesc metodele si constantele din toate superinterfetele si se pot adauga alte metode si interfete.

•O clasa care implementeaza o asemenea interfata trebuie sa implementeze toate metodele specificate in interfata, precum si toate metodele mostenite din toate super-interfetele.

Cum era mai bine sa implementam aplicatia precedenta?

Applet-urile Java (numele ar putea fi tradus prin "mici aplicatii")

Sunt programe destinate a fi rulate de catre browsere Internet cu Plug-in Java (jre) sau de catre aplicatia standard appletviewer.exe (furnizata cu kit-ul Java).
Sunt aplicatii grafice.

Applet-urile nu au metoda main si ca atare nu pot fi rulate de catre interpretorul Java.
Ele trebuie referite din fisiere de tip HTML.
Controlul HTML specific este de forma:

```
<APPLET CODE="nume_clasa.class" WIDTH="500" HEIGHT="300">
        .....Alti parametri optionali.....
</APPLET>
```

Campurile WIDTH si HEIGHT din controlul APPLET precizeaza dimensiunile in pixeli ale ferestrei grafice in care va fi rulat applet-ul.
Campul CODE precizeaza sursa compilata a applet-ului, cu observatia importanta ca aceasta sursa poate fi luata de oriunde: Internet/structura locala de fisiere. Am putea scrie:

```
<APPLET CODE="http://www.ceva.com/AppletDir/nume_clasa.class" WIDTH="500“ HEIGHT="300">
</APPLET>
```

Daca se precizeaza doar numele clasei, se presupune ca fisierul .class este local, in acelasi director cu fisierul HTML.

Clasa Applet din pachetul java.Applet este derivata din clasa Panel

java.applet

Class Applet

java.lang.Object
    java.awt.Component
        java.awt.Container
            java.awt.Panel
                java.applet.Applet

All Implemented Interfaces:

ImageObserver, MenuContainer, Serializable, Accessible

Panel – este cea mai simpla clasa Container care permite adaugarea la contextul grafic de elemente de tip Component (obiect grafic care poate fi afisat si cu care se poate interactiona): Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextComponent.

De la aceasta clasa sunt mostenite metodele:

- paint(Graphics g)
- action(Event evt, Object what)
- mouseDown(Event evt, int x, int y) , etc

care permit afisarea de elemente grafice si interactiunea cu acestea.

Clasa Applet mosteneste un grup de metode de tip event-handling din clasa Container: processKeyEvent si processMouseEvent pentru tratarea diferitelor tipuri de evenimente.

Documentatie: http://docs.oracle.com/javase/tutorial/deployment/applet/developingApplet.html
http://docs.oracle.com/javase/tutorial/uiswing/index.html
http://docs.oracle.com/javase/tutorial/2d/index.html

Clasa Applet pune la dispozitie un cadru de executie al appletului printr-o serie de metode care se apeleaza in punctele critice ale ciclului de viata al aplicatiei.

Metoda init

Este folosita la initializare, seamana cu un constructor (din punct de vedere al codului continut; initializare atribute). Metoda trebuie sa fie scurta, pentru ca aplicatia sa se incarce repede.

Metoda start

Toate aplicatiile applet care fac diverse actiuni (care nu necesita interactiunii cu utilizatorul) trebuie sa suprascrie aceasta metoda; ea porneste executia aplicatiei.

Metoda stop

In general aceasta metoda trebuie suprascrisa daca s-a suprascris metoda de start. Metoda stop suspenda executia appletului, astfel incat acesta sa nu consume resurse daca utilizatorul nu il vizualizeaza (ex: un applet care afisaza o animatie).

Metoda destroy

Multe aplicatii applet nu trebuie sa suprascrie aceasta metoda deoarece metoda stop este suficienta pentru a incheia executia. Metoda destoy este necesara pentru aplicatiile care trebuie sa elibereze resurse aditionale.

Sa consideram fisierul App1.java, cu urmatorul continut:

```
import java.awt.*;
import java.applet.*;
public class App1 extends Applet {
        int x,y;
        public void init() {
                x=50; y=50;
        }
        public void paint(Graphics g) { //g este contextul grafic pe care se face desenarea
                g.drawString("Hello World!", x, y );
        }
}
```

Instructiunile import precizeaza ca vom folosi pachetele java.applet si java.awt (Abstract Window Toolkit). Forma cu * ne permite sa ne referim doar la ultima parte a numelui entitatilor. De exemplu, numele complet al clasei predefinite Applet este java.applet.Applet.

Definitia clasei App1 (numele ei coincide cu cel al fisierului sursa) spune ca e o clasa derivata a clasei Applet. Metoda cu nume predefinit paint afiseaza applet-ul respectiv in fereastra grafica. Aceasta metoda are un parametru un obiect de tip Graphics (context grafic). Clasa Graphics contine o serie de metode de prelucrare grafica (desenare etc.). Intre ele, metoda drawString, care afiseaza un text la nivel grafic si care are trei parametri: un sir de caractere (textul care se afiseaza) si coordonatele (in pixeli fata de coltul din stanga-sus al ferestrei) pozitiei in care se face afisarea.

Compilarea acestui applet se face in modul standard.

Pentru a putea vizualiza acest applet trebuie sa creem un fisier HTML, de exemplu App1.html (nu e obligatoriu sa aiba acelasi nume cu applet-ul), cu urmatorul continut (cel putin):

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="000000">
<CENTER>
<APPLET
        code    = "App1.class"
        width   = "500"
        height  = "300"
        >
</APPLET>
</CENTER>
</BODY>
</HTML>
```

Applet-urile pot avea parametri - variabile de tip String definite in fisierul HTML si care pot fi citite in codul Java.
In exemplul de mai jos, dorim ca x-ul si y-ul unde se afiseaza String-ul "Hello World!" sa fie variabile si sa poata sa fie specificate din HTML.

```
import java.applet.*;
import java.awt.*;

public class App2 extends Applet {
        public int x1, y1;
        public void paint (Graphics g) {
                x1 = Integer.parseInt(getParameter("x"));
                y1 = Integer.parseInt(getParameter("y"));
                g.drawString("Hello World!", x1, y1);
        }
}
```

In codul appletului, citim parametrii cu metoda getParameter, care are ca argument un String ce contine numele parametrului. Valoarea parametrului se obtine tot ca String
Trebuie sa convertim acest String la int pentru ca metoda drawString presupune coordonate intregi.
Aceasta conversie se face cu metoda statica parseInt a clasei Integer.
Prototipul metodei parseInt este: **public static int parseInt(String s);**

Fisierul HTML corespunzator (App2.html) este:

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="000000">
<CENTER>
<APPLET  code= "App2.class"  width= "500“ height= "300"  >
                        <PARAM NAME="x" VALUE="20">
                        <PARAM NAME="y" VALUE="20">
</APPLET>
</BODY>
</HTML>
</APPLET>
</CENTER>
</BODY>
</HTML>
```

Observam sintaxa simpla prin care putem defini parametri: trebuie sa le dam un nume si o valoare, ambele de tip text.

In continuare e prezentat un applet ceva mai elaborat, care ilustreaza cateva elemente GUI (Graphical User Interface): butoane si liste de selectie. Fisierul Java este app3.java :

```
import java.applet.*;
import java.awt.*;

public class app3 extends Applet {
        private int last_x = 0, last_y = 0; //coordonatele anterioare mouse
        private Color current_color = Color.black; //culoare curenta pentru desenare
        private Button clear_button; //buton stergere desen
        private Choice color_choices; //lista pentru selectia culorii de desenare

        // Metoda de initializare a applet-ului
        public void init() {
                // Seteaza culoarea de fond
                 this.setBackground(Color.white);

                 // Creeaza un buton avand scris pe el textul Clear si il adauga in applet
                 clear_button = new Button("Clear");
                 // butonul are si el culori
                 clear_button.setForeground(Color.black);
                 clear_button.setBackground(Color.lightGray);
                 this.add(clear_button);
                 //este adaugat butonul in Panel
```

```
        // Creeaza o lista de selectie pentru culori
        // Seteaza culorile-optiunile din lista si adauga o eticheta
        color_choices = new Choice();
        color_choices.addItem("black");
        color_choices.addItem("red");
        color_choices.addItem("yellow");
        color_choices.addItem("blue");
        color_choices.addItem("green");
        color_choices.setForeground(Color.black);
        color_choices.setBackground(Color.lightGray);
        this.add(new Label("Color: "));
        this.add(color_choices);
        //este adaugata lista in panel
}

// Metoda apelata atunci cand utilizatorul face click pentru a incepe un desen
//mostenita din Component; mediul Java apeleaza functia la aparitia
//evenimentului
public boolean mouseDown(Event e, int x, int y)
{
        last_x = x; last_y = y; //retine coordonatele mouse
        return true;
}
```

```
// Metoda apelata atunci cand utilizatorul deplaseaza mouse-ul cu butonul
//din stanga apasat
public boolean mouseDrag(Event e, int x, int y)
{
        Graphics g = this.getGraphics();
        g.setColor(current_color);
        g.drawLine(last_x, last_y, x, y);
        last_x = x;
        last_y = y;
        return true;
}

// Metoda apelata atunci cand utilizatorul apasa butonul Clear sau
// selecteaza o culoare; trateaza actiuni asupra butoanelor,listelor,etc
public boolean action(Event e, Object arg)
{       // Daca s-a apasat butonul 'Clear'
        if (e.target == clear_button) {
                Graphics g = this.getGraphics();
                // Obtine limitele ferestrei ca dreptunghi
                Rectangle r = this.bounds();
                // Stergerea se face prin umplere cu culoarea de fond
                g.setColor(this.getBackground());
                g.fillRect(r.x, r.y, r.width, r.height);
                return true;}
```

```
                    // Daca nu, trateaza optiunile de culoare
                    else if (e.target == color_choices) {
                            if (arg.equals("black")) current_color = Color.black;
                            else if (arg.equals("red")) current_color = Color.red;
                            else if (arg.equals("green")) current_color = Color.green;
                            else if (arg.equals("blue")) current_color = Color.blue;
                        else if (arg.equals("yellow")) current_color = Color.yellow;
                        return true;
                    }
                    // Altfel, lasam superclasa Applet sa trateze evenimentul
                    else
                            return super.action(e, arg);
        }
}
```

Fisierul app3.html este obisnuit:

```
<HTML>
<BODY>
        <APPLET code="app3.class" width=400 height=300>
        </APPLET>
</BODY>
</HTML>
```

Acest applet creeaza un instrument de desenare (pen) cu ajutorul mouse-ului. Tinem mouse-ul apasat si il miscam in fereastra, iar el va lasa o urma. Dorim sa specificam culoarea de desenare printr-o lista de selectie si sa putem sterge fereastra prin actionarea unui buton.

Clasa are variabilele private last_x, last_y (coordonatele curente din care se deseneaza o linie), current_color (culoarea curenta), clear_button (buton de stergere) si color_choices (lista de selectie). Observati tipurile Button si Choice care implementeaza aceste elemente de interfata grafica.

Avem aici o alta structura de applet, utilizata in cazul cand acesta trebuie sa raspunda dinamic la comenzi utilizatori. Sa observam ca daca utilizatorul nu face nimic, applet-ul nu face nici el nimic. In cazul App2.java, applet-ul desena un text si gata.
Ca atare, vom avea o metoda init (initializarea applet-ului) care va afisa elementele de interfata. Aceasta metoda este apelata de browser sau appletviewer.

In cazul de fata, se fixeaza culoarea de fond a ferestrei (observati constanta Color.white, adica variabila static final white din clasa Color), se creeaza butonul si lista de optiuni si se adauga (se afiseaza) in applet.
Metodele utilizate sunt usor de inteles: color_choices.addItem("red") va adauga optiunea "black" in lista de selectie color_choices. La fel, this.add va adauga un element de interfata la applet.
De observat ca putem seta culoarea de fond si textul de pe buton, cu metodele setForeground si setBackground ale obiectului de tip Button.

Metodele mouseDown si mouseDrag (cu nume predefinit), vor fi apelate atunci cand utilizatorul apasa butonul stang al mouse-ului, respectiv misca mouse-ul pe ecran cu butonul stang tinut apasat. Avem aici un stil de organizare tipic aplicatiilor interactive grafice. Sintetic, el ar fi caracterizat prin "don't call me - I'll call you". Ideea este ca utilizatorul isi scrie functii cu nume predefinite, care definesc actiunile care trebuie efectuate in urma unor evenimente (mouse apasat, deplasata etc.). Sistemul de operare (in cazul de fata mediul Java) este cel care apeleaza aceste functii. Acest mod de organizare este exact invers decat cel cu care eram obisnuiti pana acum: in general, codul utilizator apeleaza functii de sistem. La aplicatiile GUI este exact pe dos.

Metoda mouseDown memoreaza pozitia curenta a mouse-ului (primita in x si y) in last_x si last_y. Metoda mouseDown primeste noua pozitie (x si y) si traseaza o linie (cu drawLine) intre vechea pozitie si cea curenta, dupa care pozitia noua devine pozitie veche. Pentru a desena o linie avem nevoie de contextul grafic al applet-ului, pe care il obtinem prin this.getGraphic(). Observati ca la app1.java contextul grafic era parametru al metodei paint.

Pana acum am tratat evenimentele de apasare si deplasare ale mouse-ului. Mai avem de tratat actiunea utilizatorului (prin tastatura sau mouse) asupra elementelor de interfata (Button si Choice). Aceasta tratare se face in metoda cu nume predefinit action, care primeste un eveniment (Event) si un argument asociat evenimentului. Observati ca argumentul este de tipul cel mai general posibil (Object - radacina tuturor claselor Java).

Pentru a vedea care element de interfata a fost actionat, se foloseste campul target al evenimentului. Acesta contine o referinta la obiectul actionat. Se testeaza pur si simplu daca e.target este clear_button sau color_choices.

Pentru stergere se foloseste metoda fillRect (umple dreptunghi) cu culoarea de fond a ferestrei (obtinuta prin this.getBackground()). Pentru a gasi dreptunghiul ferestrei se utilizeaza metoda getBounds (obtine marginile) care intoarce un obiect de tip Rectangle (dreptunghi). Acesta are campurile x, y, width si height (coltul din stanga-sus, latimea si inaltimea). Exact aceste valori trebuie date functiei fillRect. Evident, si aici avem nevoie de contextul grafic.

In cazul actionarii listei de selectie, se primeste in obiectul arg optiunea selectata. Aceasta optiune este de tip text si coincide cu una din optiunile care au fost aduagate la crearea listei. Se testeaza pur si simplu egalitatea dintre aceste siruri de caractere.

Aici trebuie remarcat un lucru important: String-urile sunt obiecte, deci a testa in forma:

```
if ( arg == "green")
```

inseamna o comparatie intre doua referinte: arg si o referinta la sirul constant "green". Aceste referinte nu vor fi niciodata egale. Noi dorim sa comparam continutul celor doua siruri, ceea ce se realizeaza prin metoda standard equals a clasei Object:

```
if ( arg.equals("green') )
```

Functie de optiunea selectata, se actualizeaza culoarea curenta de desenare.

In fine, daca tinta (target) evenimentului nu este nici unul din elementele de interfata definite de noi, transmitem acest eveniment superclasei Applet (clasa de baza a clasei app3).

Aceasta pentru ca pot exista multe alte evenimente care trebuiesc procesate: miscarea ferestrei pe ecran, redimensionarea sa, accesul la meniurile ferestrei, inchiderea ferestrei. Toate acestea vor fi prelucrate de clasa de baza Applet.

Ce nu am acoperit in aceste cursuri:

- **Colectii, Exceptii, Sabloane, Metode finalize etc**

Tutorial rapid:
http://www.tutorialspoint.com/java/index.htm