INTERFEŢE

Noţiunea de interfaţă

Aşa cum s-a menţionat anterior, este posibil să declarăm clase abstracte în care toate metodele sunt abstracte. Acest lucru poate fi realizat şi prin intermediul *interfeţelor*. În plus, ele reprezintă mecanismul propus de Java pentru tratarea problemelor ridicate de moştenirea multiplă.

Interfeţele reprezintă o modalitate de a declara un tip constând numai din constante şi din metode abstracte. Ca sintaxă, o interfaţă este asemănătoare unei clase, cu deosebire că în loc de **`class`** trebuie precizat **`interface`**, iar metodele nu au corp, acesta fiind înlocuit cu `';'`.

Putem spune că interfeţele reprezintă numai **proiecte**, pe când clasele abstracte sunt o combinaţie de **proiecte şi implementări**.

Ca şi în cazul claselor abstracte, este evident că **nu pot fi create obiecte de tipul unei interfeţe.**

Câmpurile unei interfeţe au în mod implicit modificatorii **`static`** şi **`final`**, deci sunt constante.

Metodele interfeţelor sunt totdeauna publice şi au în mod implicit modificatorul **`abstract`**. În plus ele nu pot fi statice, deoarece fiind abstracte, nu pot fi specifice claselor.

Orice interfaţă este gândită pentru a fi ulterior *implementată* de o clasă, în care metodele interfeţei să fie redefinite, adică să fie specificate acţiunile ce trebuie întreprinse. Faptul că o clasă `C` implementează o interfaţă `I` trebuie specificat prin inserarea informaţiei **`implements`** `I` în antetul clasei.

Exemple de utilizare a interfeţelor

1. Proiecte de clase – vezi curs

2. Interfeţele permit transmiterea metodelor ca parametri

```
interface Functie{
      double f(double x);
}
class F1 implements Functie{
      double a,b,c;
      F1(int a1,int b1, int c1){ a=a1;b=b1;c=c1; }
      public double f(double x){ return a*x*x+b*x+c; }
}
```

```
class F2 implements Functie{
      public double f(double x){return Math.sin(x); }
}
class MainF{
      static double f(Functie ob, double x){
            return ob.f(x);
      }
      public static void main(String arg[]){
            System.out.println(f(new F1(1,1,1),2));
            System.out.printf("%.2f",f(new F2(),Math.PI/2));
      }
}
```

Precizări:

- dacă o clasă implementează doar unele din metodele unei interfeţe, atunci ea trebuie declarată cu **abstract**;

- o clasă poate implementa oricâte interfeţe, dar poate extinde o singură clasă.

- o interfaţă poate extinde oricâte interfeţe. **Facilitatea ca interfeţele şi clasele să poată extinde oricâte interfeţe reprezintă modalitatea prin care Java rezolvă problema moştenirii multiple.**

- dacă o clasă extinde o altă clasă şi implementează una sau mai multe interfeţe, atunci trebuie anunţată întâi extinderea şi apoi implementarea, ca în exemplul următor:

```
class C extends B implements I1,I2,I3;
```

Exemplu

```
interface A {
  int x=1;
  void scrie();
}
interface B extends A {
  int x=2;
  void scrieB();
}
class C {
  int x=3;
  public void scrie() { System.out.println(A.x+" "+B.x+" "+x); }
}
class D extends C implements B {
  int x=4;
  void printx(){ System.out.println(x+" "+super.x); }
  public void scrieB(){System.out.println("B"); }
}
```

```
class Interfete2 {
  public static void main(String[] w) {
    D ob = new D();
    ob.scrie();
    System.out.println(ob.x);
    ob.printx();
    ob.scrieB();
    A ob1 = new D();
    ob1.scrie();
  }
}
```

produce la ieşire:

```
1 2 3
4
4 3
B
1 2 3
```

Interfeţe din Java

1. Interfaţa Comparable - „ordinea naturală" a obiectelor

Interfaţa `Comparable<T>` este din pachetul java.lang şi are o singură metodă

```
int compareTo(T o)
```

Exemplu- vezi curs

```
import java.util.*;
class Persoana implements Comparable<Persoana>{
      String nume;
      int varsta;
      public int compareTo(Persoana persoana){
            return varsta-persoana.varsta;
      }
}
```

Recomandare: Ordinea naturală pentru o clasă `C` trebuie să fie ***consistentă faţă de metoda equals*** adică `e1.compareTo(e2) == 0` ⇔ `e1.equals(e2)` pentru orice două obiecte `e1` şi `e2` de tip `C`.

2. Interfaţa Iterator

- **iterator** = un obiect care permite parcurgerea tuturor elementelor unei colecţii, unul câte unul, indiferent de implementare (având ca tip o clasă care implementează interfaţa `Iterator`)
- pentru a accesa elementele cu ajutorul unui iterator, clasa trebuie să aibă o metodă care furnizează iteratorul, mai exact să implementeze interfaţa `Iterable`
- Interfaţa `Iterator<E>` din pachetul `java.util` are metodele

```
boolean hasNext()
E next() //throws NoSuchElementException
void remove()//throws UnsupportedOperationException,
IllegalStateException
```

- Interfaţa `Iterable<T>` din pachetul `java.util` are metoda

```
Iterator<T> iterator()
```

Exemplu În exemplul de mai jos este implementată o colecţie de caractere iterabilă, memorată intern ca **`String`** (exemplul fiind didactic, pentru a ilustra ideea de iterator). Pentru a simplifica accesul clasei **`IteratorSir`** la membrii clasei **`Sir`** clasa **`IteratorSir`** este clasă internă a clasei **`Sir`**.

```
import java.util.*;
class Sir implements Iterable<Character> {
      private String s;
      Sir(){}
      Sir(String s1){ s=s1 ;}
      Sir(char[] s1){ s=new String(s1);}
      public Iterator<Character> iterator(){
            return new IteratorSir();
      }

      class IteratorSir implements Iterator<Character>{
            private int curent=0;
            public boolean hasNext(){
                  return curent<s.length();
            }
            public Character next(){
                  if (curent==s.length())
                        throw new NoSuchElementException();
                  char c= s.charAt(curent);
                  curent++;
                  return c;
            }
```

```
            public void remove() {
                throw new UnsupportedOperationException();
            }
      }
}
class ExpIterat{
      public static void main(String aa[]){
            char vc[]={'x','y','z'};
            Sir sc=new Sir(vc);
            for(Character c:sc)
                  System.out.println(c);
            Sir s=new Sir("abcdef");
            for(Character c:s)
                  System.out.println(c);
            for(char c:s)//merge si cu char, unboxing
                  System.out.print(c);
            System.out.println();
            Iterator<Character> it=s.iterator();
            while(it.hasNext())
                  System.out.println(it.next());
            //System.out.println(it.next());    //NoSuchElementException
      }
}
```

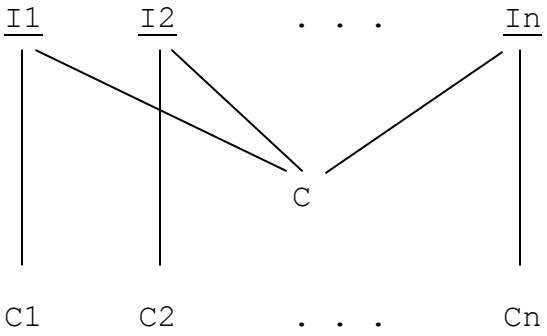
Observaţie. Puteam scrie clasa **`IteratorSir`** ca o clasă exterioară clasei **`Sir`**, dar pentru ca aceasta să poată accesa membrii clasei **`Sir`** trebuie să adăugăm în clasa **`IteratorSir`** un câmp de tip **`Sir`** şi un constructor care primeşte ca parametru un **`Sir`**, iar metoda `iterator`() va fi:

```
      public Iterator<Character> iterator(){
            return new IteratorSir(this);
      }
```

- **Rezolvarea în Java a problemei moştenirii multiple**

Să presupunem că plecând de la clasele `C1, C2, ..., Cn` dorim să construim o nouă clasă care să moştenească unele dintre metodele lor. Java permite doar moştenire simplă, deci va fi necesar să apelăm la interfeţe.

Modalitatea de rezolvare a problemei moştenirii multiple pe care o prezentăm în continuare este independentă de numărul de clase moştenite. Vom folosi următoarea structură de interfeţe şi clase:



În această structură clasele `C1, C2, ..., Cn` implementează respectiv metodele anunţate în interfeţele `I1, I2, ..., In`, iar clasa `C` implementează toate interfeţele `I1, I2, ..., In`. Este suficient să prezentăm modul în care clasa `C` moşteneşte implementările din `C1` ale metodelor anunţate în `I1`, lucrurile decurgând analog pentru celelalte interfeţe şi clase pe care le implementează. De aceea *în continuare vom presupune că* `C` *trebuie să extindă doar clasa* `C1`.

În clasa `C` vom declara şi crea un obiect `ob1` de tipul `C1` (se presupune că în clasa `C` se ştie ce implementare a interfeţei `I1` trebuie folosită). Atunci pentru fiecare metodă `met` implementată de `C1` introducem în clasa `C` metoda `met` cu aceeaşi signatură şi având una dintre formele:

tip `met(...) { return ob1.met(...); }`

`void met(...) { ob1.met(...); }`

după cum metoda întoarce sau nu o valoare.

Exemplu

```
interface X {
  void x1();
  int x2();
}

class CX implements X {
  public void x1() { System.out.print("x1 "); }
  public int x2() { return 1; }
}

class C implements X {
  X obX= new CX(); // Clasa C "stie" ca trebuie sa foloseasca
                   // implementarea CX a interfetei X
  public void x1() { obX.x1(); }
  public int x2() { return obX.x2(); }
}
```

Interfeţe grafice

Facilităţi Java de lucru cu interfeţe apar în `java.awt` (*Abstract Windowing Toolkit*).
Există trei aspecte care trebuie luate în considerare:

1) crearea şi afişarea pe ecran a unei suprafeţe grafice pe care urmează a fi plasate diferite obiecte grafice; pentru suprafaţa grafică trebuie stabilită o configuraţie iniţială (un şablon), constând din dimensiunile ei, unele obiecte grafice plasate de la început pe ea etc.;
2) o modalitate de poziţionarea a obiectelor pe suprafaţa grafică;
3) un mecanism prin care programul să răspundă la diferite acţiuni executate asupra obiectelor grafice, conform dorinţelor programatorului.

Exemplul 1 Sunt create un buton şi un câmp de text. Acţiunile asupra lor nu au nici un efect, iar fereastra nu poate fi închisă.

```
import java.awt.*;
public class Ex1 {
  public static void main(String[] args) {
    F f = new F(" Ex1 ");
    f.setSize(300,100);//al doilea parametru e inaltimea
    f.setLocation(200,200);//in ce punct apare fereastra
    f.setVisible(true);
  }
}
class F extends Frame {
  //static final long serialVersionUID=1;
  Button btn; TextField txt;
  F(String s) {
    setTitle(s);
    setLayout(new FlowLayout());
    txt = new TextField(6);   add(txt);
    btn = new Button("OK");  add(btn);
  }
}
```

Principalele suprafeţe grafice corespund următoarelor clase din `java.awt`:

```
Component (clasă abstractă)
   Container
      Window
         Frame
         Dialog
      ScrollPane
      Panel
         Applet
```

- Clasa abstractă `Component`, care este rădăcina arborelui claselor din AWT, defineşte o interfaţă generală pentru toate componentele grafice. Obiectele de tip `Component` se numesc *componente*.
- Un *container* (obiect de tip `Container`) este o componentă care poate include alte componente AWT, folosind de obicei o structură de listă.
- *fereastă AWT* (obiect de tipul `Window`) este o fereastră fără margini şi fără bară de meniuri.
- Un *cadru* (obiect de tip `Frame`) este o ferestră AWT cu margini, titlu şi bară de meniuri.
- *O fereastă de dialog* (obiect de tipul `Dialog`) extinde o fereastră AWT prin adăugarea de margini şi a unui titlu; ea este folosită în principal pentru a prelua informaţii de la utilizator.
- Un *panou* (obiect de tipul `Panel`) este folosit pentru a grupa mai multe componente.

Crearea unei suprafeţe grafice nu implică afişarea ei. De aceea am invocat metoda `setVisible` cu argumentul `true`. Pentru setarea dimensiunilor cadrului (în pixeli), am invocat metoda statică `setSize`.

Pentru plasarea componentelor în container, trebuie folosit un *gestionar de poziţionare* (*layout manager)*. **Fiecărui container îi este asociat un gestionar de poziţionare implicit** (pentru cadre este folosit `BorderLayout`).Vom folosi în continuare doar `FlowLayout`, care plasează componentele în container de la stânga la dreapta şi de sus în jos. Gestionarul de poziţionare a fost specificat prin invocarea metodei **`setLayout`** din clasa `Container`.

- **Controale grafice (componente elementare)**

Controalele grafice permit interacţiunea cu utilizatorul. Ele corespond claselor ce apar subliniate în următoarea ierarhie:

```
Component
   Button              buton
   Canvas              suprafaţă de desenare
   Checkbox            căsuţă de opţiuni
   Choice              listă de opţiuni cu alegere unică
   Label               etichetă
   List                listă de opţiuni cu alegere multiplă
   Scrollbar           bară de defilare
   TextComponent
      TextArea         arie de text
      TextField        câmp de text
```

Controalele grafice pot fi adăugate containerelor; adăugarea se face conform regulilor stabilite de gestionarul de poziţionare ales. Fie `c` un container şi `ob` un control grafic. Adăugarea lui `ob` la `c` se face astfel:

```
c.add(ob);
```

Schema generală de lucru cu AWT

Schema urmează aşa-numita *programare orientată pe evenimente*.

- **Evenimente**

Dorim ca la o anumită acţiune asupra unei anumite componente, să putem preciza o acţiune specifică.
Când acţionăm asupra unei componente, apare un *eveniment*. **Un eveniment este un obiect de tipul unei anumite clase.** Evenimentul are un *`tip`*, care este o clasă *`tip`*`Event` din pachetul `java.awt.event`.

Iată o scurtă listă de evenimente:

Tip	Un eveniment de acest tip apare dacă acţionăm asupra:
`Action`	`Button, List, TextField`
`Adjustment`	`Scrollbar`
`Text`	`TextArea, TextField`
`Mouse`	`Mouse`: click, apăsare, eliberare
`Window`	`Window`: închidere, minimizare/maximizare

- Pentru a specifica o acţiune la apariţia unui eveniment, trebuie mai întâi să îi asociem un *gestionar* (*listener*, *handler*) `h`. Un handler (gestionar) este un obiect al unei clase ce implementează interfaţa *`tip`*`Listener` sau extinde clasa *`tip`*`Adapter` (numai pentru tipurile `Mouse` şi `Window`), unde *`tip`* este tipul acţiunii asupra lui `c` pe care dorim să o controlăm.

Fie de exemplu `c` un control grafic sau o fereastră. Putem să îi asociem un gestionar `h` (mai spunem că `h` este înregistrat ca gestionar pentru `c`) astfel:

```
c.addtipListener(h);
```

- Pentru fiecare tip de acţiune asupra unui control grafic, există o metodă anunţată în interfaţa *tip*Listener sau în clasa *tip*Adapter, ambele din pachetul java.awt.event. De exemplu interfaţa MouseListener anunţă metode separate pentru click, apăsare şi eliberare.

Toate metodele din clasele *tip*Adapter specifică acţiunea vidă.

În general, metodele din interfeţele *tip*Listener şi din clasele *tip*Adapter au un parameter de tipul *tip*Event. Acesta foloseşte pentru a identifica controlul grafic asupra căruia s-a acţionat (de exemplu două butoane pot folosi acelaşi handler).

Exemplul 2 În plus faţă de exemplul anterior, când apăsăm pe buton, apare mesajul "O.K." în câmpul de text.

Varianta 1 – Gestionarul este clasă separată de clasa pentru fereastră

```
import java.awt.*;
import java.awt.event.*;

public class Ex2 {
  public static void main(String[] args) {
    F f = new F(" Ex2 ");
    f.setSize(300,100); f.setLocation(200,200);
    f.setVisible(true);
  }
}

class F extends Frame {
  Button btn; TextField txt;
  F(String s) {
    setTitle(s);
    setLayout(new FlowLayout());
    txt = new TextField(6);  add(txt);
    btn = new Button("OK");  add(btn);
    BHandler bh = new BHandler(this);
    btn.addActionListener(bh);
  }
}
class BHandler implements ActionListener {
  F f;
  BHandler(F f) { this.f=f; }
  public void actionPerformed(ActionEvent e) {
    f.txt.setText("O.K.");
  }
}
```

Observaţii:

- singura metodă anunţată în interfaţa `ActionListener` este **`actionPerformed`**;
- în continuare cadrul nu poate fi închis;
- este permis să specificăm drept handler chiar clasa curentă, dar este recomandat să separăm aplicaţia de implementare.

Varianta 2 – Gestionarul este clasă internă a clasei pentru fereastră

```
class F extends Frame {
  Button btn; TextField txt;
  F(String s) {
    setTitle(s);
    setLayout(new FlowLayout());
    txt = new TextField(6);  add(txt);
    btn = new Button("OK");  add(btn);
    BHandler bh = new BHandler();
    btn.addActionListener(bh);
  }
  class BHandler implements ActionListener {//clasa interna
     public void actionPerformed(ActionEvent e) {
      txt.setText("O.K.");
    }
  }
}
```

Varianta 3 – Clasa pentru fereastră este şi gestionar

```
class F extends Frame implements ActionListener {
  Button btn; TextField txt;
  F(String s) {
    setTitle(s);
    setLayout(new FlowLayout());
    txt = new TextField(6);  add(txt);
    btn = new Button("OK");  add(btn);
    btn.addActionListener(this);
  }
  public void actionPerformed(ActionEvent e) {
    txt.setText("O.K.");
  }
}
```

Exemplul 3 Adăugăm în exemplul anterior şi un gestionar pentru a putea închide fereastra. Pentru închiderea ferestrei vom folosi o nouă clasă care implementează interfaţa WindowListener (!!implementând toate metodele acesteia) sau extinde clasa `WindowAdapter` :

```
class WHandler extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
}
```

Asociem ferestrei acest gestionar adăugând în constructorul clasei `F` următorul cod

```
WHandler wh = new WHandler();
addWindowListener(wh);
```

Observaţie. Pentru închiderea ferestrei puteam apela la clase anonime. Astfel, clasa `WHandler` nu mai era necesara, ci puteam adăuga direct în constructorul clasei `F` următorul cod

```
addWindowListener(
  new WindowAdapter() {
    public void windowClosing(WindowEvent e)
      { System.exit(0); }
  }
); //clasa anonima
```

Observaţie. Este posibil să ataşăm mai mulţi gestionari unui aceluiaşi control grafic; în acest caz, declanşarea evenimentului este comunicată tuturor gestionarilor. Aceasta se întâmplă de obicei dacă o bună parte dintre acţiunile ce trebuie întreprinse sunt comune. Pentru acţiunile specifice, trebuie însă să diferenţiem între controale grafice.
Dacă controalele grafice sunt de tipuri diferite, putem folosi metoda:

```
public Object getSource()
```

pentru a identifica sursa.
Dacă controalele grafice sunt de acelaşi tip, putem folosi metoda:

```
public String getActionCommand()
```

care întoarce eticheta sursei.

De asemenea, acelaşi handler poate fi asociat mai multor controale grafice.

- **Suprafeţe de desenare. Clasele `Canvas` şi `Graphics`**

Suprafeţele grafice oferă facilitatea de control la nivel de pixel. Pe o astfel de suprafaţă se pot desena, folosind metode ale clasei `Graphics`, segmente de dreaptă, dreptunghiuri, ovale etc.; la desenare poate fi aleasă o culoare. De asemenea contururile închise pot fi "umplute" cu o culoare. Pe o suprafaţă de desenare mai pot fi plasate şi imagini, pot fi create propriile butoane etc.

Clasa `Canvas` este complexă, nu atât direct prin metodele sale, ci mai ales prin posibilitatea de a folosi alte clase ca de exemplu cele pentru grafică bi- şi tridimensională. Vom reduce însă prezentarea la câteva facilităţi mai des folosite.

Modul standard de creare a unei suprafeţe grafice constă în extinderea clasei `Canvas`, redefinind metoda `paint`, ce afişează componenta. Modificările ulterioare devin vizibile prin invocarea metodei `repaint`.

O suprafaţă grafică reprezintă un dreptunghi de pixeli, cu colţul din stânga sus având coordonatele `(0,0)`. Este folosit un *context de desenare*; nu vom intra în amănunte, ci vom folosi contextul de desenare standard disponibil.

Pentru clasa `Canvas` precizăm doar următoarele elemente:

```
public class Canvas extends Component
  public Canvas()
  public paint(Graphics g)
```

cu menţiunea că invocarea metodei `paint` pune la dispoziţie contextul de desenare standard `g`. Faptul că este extinsă clasa `Component` permite captarea de evenimente specifice acestei clase, ca de exemplu `Key`, `Mouse`, `MouseMotion`.

Prezentăm în continuare câteva elemente legate de clasa `Graphics`. Atât clasa, cât şi metodele sale, au modificatorii `public` şi `abstract`. Menţionăm că un context de desenare are ataşate câteva informaţii curente, printre care culoarea curentă de desenare şi fontul curent. Drept urmare, de exemplu trasarea unei linii şi "umplerea" unui contur se fac folosind culoarea curentă.

```
class Graphics extends Object
  Color getColor()
  void setColor(Color c)
```

este obţinută, respectiv setată, culoarea curentă de desenare;

```
  Font getFont()
  void setFont(Font f)
```

este obţinut, respectiv setat, fontul curent;

```
  void drawLine(int x1, int y1, int x2, int y2)
```

este trasat un segment între punctele de coordonate `(x1,y1)` şi `(x2,y2)`;

`void drawPolyline(int[] x, int[] y, int n)`
este trasată o linie poligonală ce trece, în ordine, prin `n` puncte ale căror coordonate sunt specificate în tablourile `x` şi `y`;
`void drawRect(int x, int y, int w, int h)`
este trasat un dreptunghi: colţul din stânga sus are coordonatele `(x,y)`, iar lăţimea şi înălţimea sa sunt `w`, respectiv `h`;
`void fillRect(int x, int y, int w, int h)`
un dreptunghi specificat ca mai sus este "umplut" cu culoarea curentă;
`void drawOval(int x, int y, int w, int h)`
este trasată o elipsă tangentă la laturile dreptunghiului precizat;
`void fillOval(int x, int y, int w, int h)`
este umplută o elipsă tangentă la laturile dreptunghiului precizat.

Exemplu Dorim să figurăm pe ecran puncte din plan (raportate la axele de coordinate) şi să unim prin linii puncte consecutiv introduse.

```
import java.awt.*; import java.awt.event.*;
import java.io.*; import java.util.*;
class Linii {
  public static void main(String[] args) {
    F f = new F(" Desen ");
    f.setSize(600,600); f.setLocation(50,50); f.setVisible(true);
  }
}
class F extends Frame {
 F(String s) {
    setTitle(s);
    Grafic g = new Grafic(); add(g);
    addWindowListener( new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          System.exit(0);
        }
      } );
  }
}
class Grafic extends Canvas {
  public void paint(Graphics g) {
    g.translate(300,300);
    g.setColor(Color.red);
    g.drawLine(-300,0,300,0);
    g.drawLine(0,-300,0,300);
    g.setColor(Color.BLACK);
```

```
    Scanner sc=null;
    try {
        sc = new Scanner( new FileInputStream("in.txt") );
        int xa,ya,x,y;
         xa = Math.round(20*sc.nextFloat());
         ya = Math.round(20*sc.nextFloat());
         while( sc.hasNextFloat() ) {
            x = Math.round(20*sc.nextFloat());
            y = Math.round(20*sc.nextFloat());
            g.drawRect(x,y,1,1);
            g.drawLine(xa,ya,x,y);
            xa=x; ya=y;
         }
    }
    catch(FileNotFoundException e) {
      System.out.println("Fisier inexistent");
    }
  }
}
```

```
in.txt
-10 -10
-10 -4
-1 -8
-1 3
10 8
10 -5
```

Suplimentar

Exemplu Graphics2D

```
import java.awt.*;import java.awt.event.*;import java.awt.geom.*;
class Test2D extends Frame{
  Canvas2d c2d;
  Test2D(){
      super("Test 2D");
      c2d=new Canvas2d(); add(c2d);
      addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                        System.exit(0);
                  }
      });
  }
```

```
  public static void main(String ar[]){
      Test2D t=new Test2D();
      t.setSize(400,400);
      t.setLocation(100,100);
      t.setVisible(true);
  }
}
class Canvas2d extends Canvas{
  public void paint(Graphics g){
    Graphics2D g2=(Graphics2D)g;
    Dimension d=getSize();
    int x1,y1,x2,y2,ctrlx1, ctrly1, ctrlx2, ctrly2;
    x1=10; y1=10; x2=d.width-10; y2=10;
    Line2D l1=new Line2D.Double(x1,y1,x2,y2);
    g2.draw(l1);
    Line2D  l2=new  Line2D.Double(new  Point2D.Double(x1,y1+5),  new
Point2D.Double(x2,y2+5));
    g2.draw(l2);
    x1=50; y1=50; x2=d.width-50; y2=50;
    ctrlx1=100; ctrly1=120;

    QuadCurve2D q = new QuadCurve2D.Float();
    q.setCurve(x1, y1, ctrlx1, ctrly1, x2, y2);
    g2.draw(q);
    x1=50; y1=150;
    x2=d.width-50; y2=150;
    ctrlx1=80; ctrly1=60;
    ctrlx2  =100; ctrly2=280;

    CubicCurve2D c = new CubicCurve2D.Double();
    c.setCurve(x1, y1, ctrlx1, ctrly1, ctrlx2, ctrly2, x2, y2);
    g2.draw(c);

    GeneralPath arbShape =  new GeneralPath();
    arbShape.moveTo (110,250); arbShape.lineTo(200,250);
    arbShape.lineTo(160,300); arbShape.moveTo(150,300);
    arbShape.lineTo(110,250); arbShape.closePath();
    g2.draw(arbShape);
  }
}
```