Programare orientată pe obiecte

1. Applet-uri
2. Colecţiile Java

Ce este un applet Java?

- Applet-urile sunt pur şi simplu programe Java care rulează în browsere de web.
- Au făcut mare vâlvă pe la mijlocul anilor 90 când Netscape a fost de acord să includă JVM browserul de Web "Navigator".
- Marea promisiune – aplicaţiile să poată fi distribuite automat oricui, pe orice platformă!
- Realitatea – suportul din browsere este neuniform, au apărut limitări impuse se securitate, au apărut alte modalităţi mai simple de a face acelaşi lucru!

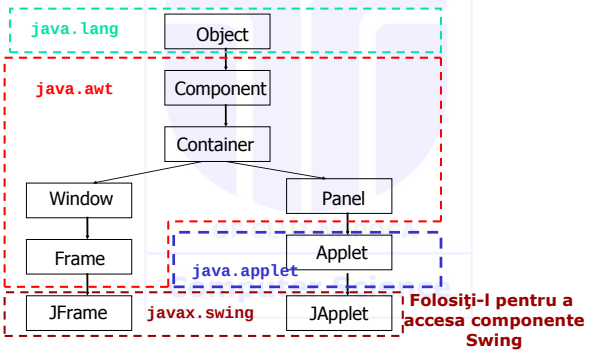
Ce este un applet Java?

- Încă util în situaţia potrivită
  - client decorat, dezvoltat
  - poate face presupuneri/avea un anumit control asupra tehnologiei client
- De asemenea, foarte util pentru înţelegerea problemelor din programarea client-server pe Web
- Altfel, programarea pe partea de server cu HTML sau scenarii ("scripting", folosind d.e. JavaScript) pe partea de client este mai avantajoasă.
- De asemenea, Java WebStart – noua alternativă
  - permite descărcarea aplicaţiilor prin browser şi rularea lor independent de browser, folosind JVM din sistemul respectiv.

Utilizarea practică a applet-urilor

- Applet-urile au utilizări serioase pe intranet-uri. Motive:
  - Intranet-urile corporaţiilor sunt sigure şi administrate -> multe din restricţiile de securitate normale pot fi relaxate.
  - Uşurinţa administrării. Un număr mare de aplicaţii foarte mici distribuite peste foarte multe maşini poate crea o problemă de administrare (d.e. actualizarea periodică) – utilizatorii mai puţin capabili le pot muta, deteriora sau şterge. Applet-urile de pe un server central, încărcate la cerere, nu au această problemă (deşi este nevoie să se întreţină browserele, plug-in-urile etc.).
  - Lărgimea de bandă mare. Reţelele interne au în general lărgimi mari de bandă – aceasta înseamnă că încărcarea repetată chiar a applet-urilor mari poate să nu fie o problemă.
- Reprezintă încă o tehnologie viabilă pentru aplicaţii care au nevoie de o interfaţă mai robustă decât aceea dintr-o pagină HTML cu JavaScript.

Arborele de descendenţă pentru Applet



Diferenţe esenţiale între applet-uri şi aplicaţiile de sine stătătoare

- Întotdeauna se creează o instanţă a applet-ului, iar constructorul său, **`init`**, şi metoda **`start`** este executată întotdeauna.
- Pentru că o instanţă de **`Applet`** este o instanţă de **`Panel`** (iar **`JApplet`** extinde **`Applet`**), se creează o componentă vizibilă, sunt tratate (potenţial) evenimentele **`awt`** etc.
- La invocarea unei aplicaţii de sine stătătoare, se execută ***numai*** **`public static void main( String[] )`** (şi codul apelat de acesta).

Metodele unui applet

- Fiecare applet are 5 metode standard :
  - `init()` — apelată de browser când applet-ul este încărcat în memorie
  - `start()` — apelată de browser pentru a porni animaţiile care se execută atunci când applet-ul este (din nou) vizibil
  - `stop()` — apelată de browser pentru a opri animaţiile în execuţie atunci când applet-ul este acoperit sau minimizat
  - `destroy()` — apelată de browser imediat înainte de a distruge applet-ul
  - `paintComponent()` — apelată de browser atunci când applet-ul este desenat sau redesenat

Metodele unui applet

- Metodele unui applet vor fi invocate întotdeauna în ordinea `init`, `start`, `stop`, `destroy` – ciclul de viaţă al appletului
- `start` şi `stop` pot fi apelate de multe ori pe durata de viaţă a unui applet
- Toate cele 5 metode sunt implementate fără conţinut în clasa `JApplet`, iar aceste metode sunt *moştenite* de toate applet-urile
- Applet-urile suprascriu *numai* metodele de care au nevoie pentru a-şi îndeplini funcţiile
- Applet-ul are o fereastră de stare (dedesubt)
  - Informează ce face applet-ul în momentul respectiv
  - Pentru a scrie in fereastra de stare folosiţi metoda `showStatus` cu un argument `String`

Crearea unui applet

- Pentru a crea un applet bazat pe Swing:
  - Creăm o subclasă a lui `JApplet` pentru a păstra componentele GUI
  - Alegem un gestionar de aranjare pentru container, dacă gestionarul implicit ((`BorderLayout`; aranjarea implicită este `FlowLayout` pentru `Applet`)) nu este acceptabil
  - Creăm componentele şi le adăugăm la *panoul de conţinut (content pane)* al containerului `JApplet`.
  - Creăm obiecte "ascultător" pentru a detecta şi a răspunde la evenimentele aşteptate de fiecare componentă GUI şi asignăm ascultătorii componentelor corespunzătoare.
  - Creăm un fişier text HTML pentru a preciza browser-ului care applet Java trebuie încărcat şi executat

Exemplu: crearea unui applet

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class FirstApplet extends JApplet {
  // Instance variables
  private int count = 0;          // Numarul de apasari
  private JButton pushButton; // Buton care se apasa
  private JLabel label;           // Eticheta
  // Initialization method
  public void init() {
    // Set the layout manager
    getContentPane().setLayout( new BorderLayout() );
    // Create a label to hold push count
    label = new JLabel("Push Count: 0");
    getContentPane().add( label, BorderLayout.NORTH );
    label.setHorizontalAlignment( label.CENTER );
    // Create a button
    pushButton = new JButton("Test Button");
    pushButton.addActionListener( new ButtonHandler(this) );
    getContentPane().add( pushButton, BorderLayout.SOUTH );
  }
  // Method to update push count
  public void updateLabel() {  label.setText( "Push Count: " + (++count) );  }
}
```

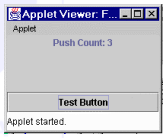
Acest applet simplu implementează `init` şi *moşteneste* toate celelalte metode fară efect

Remarcaţi că toate componentele sunt adaugate la `ContentPane` al `JApplet`

Exemplu: crearea unui applet

```
class ButtonHandler implements ActionListener
{
  private FirstApplet fa;
  // Constructor
  public ButtonHandler ( FirstApplet fa1 )
  {
    fa = fa1;
  }
  // Execute when an event occurs
  public void actionPerformed( ActionEvent e )
  {
    fa.updateLabel();
  }
}
```

Clasă ascultător pentru buton pe applet




```
<html>
  <applet code="FirstApplet.class" width=200 height=100>
  </applet>
</html>
```

cod HTML pentru lansarea applet-ului in browser

Parametri pentru applet-uri

- **Parametrii sunt stocaţi ca parte a paginii de Web care conţine applet-ul.**
- **Parametrii sunt creaţi folosind eticheta HTML `<PARAM>` şi cele două atribute ale sale: `NAME` şi `VALUE`.**
- **Pot exista mai multe etichete `<PARAM>` pentru un applet. Toate trebuie incluse între eticheta de început `<APPLET>` şi cea de sfârşit `</APPLET>`. Exemplu cu mai mulţi parametri:**

```
<APPLET CODE="ScrollingHeadline.class" HEIGHT=50 WIDTH=400>
<PARAM NAME="Headline1" VALUE="Highest marks assigned">
<PARAM NAME="Headline2" VALUE="All people took part">
</APPLET>
```

- **Atributul `NAME` se foloseşte pentru a da un nume parametrului. Atributul `VALUE` dă parametrului numit o valoare.**
- **Obţinerea parametrilor:metoda `getParameter`. D.e.**

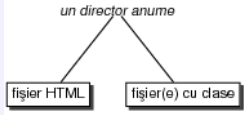
```
String display1 = getParameter("Headline1");
```

Etichete HTML pentru applet-uri

```
<APPLET
[CODEBASE = URLbazaPentruCod]
CODE = fisierApplet
[ATL = textAlternativ]
[NAME = numeInstantaApplet]
WIDTH = latimeInPixeli HEIGHT = inaltimeInPixeli
[ALIGN = alignment]
[VSPACE = spatiuVertInPixeli] [HSPACE =
  spatiuOrizInPixeli]
[<PARAM NAME = numeParametru1 VALUE = valoareParametru1>]
[<PARAM NAME = numeParametru2 VALUE = valoareParametru2>]
...
[<PARAM NAME = numeParametrun VALUE = valoareParametrun>]
[HTML care va fi afisat in absenta Java]
</APPLET>
```

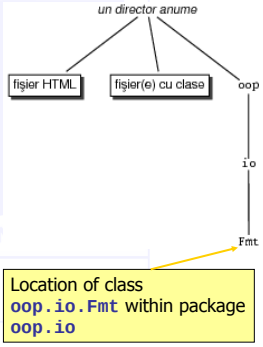

Locul în care se află fişierele clasă pentru applet-uri

- Dacă un applet foloseşte o clasă care *nu* se află într-un pachet, clasa respectivă trebuie să fie prezentă în *acelaşi director* ca şi fişierul HTML folosit pentru lansarea applet-ului
  - Acest director poate fi local sau la distanţă — nu contează
  - Fişierele clasă pot fi transferate peste reţea, aşa că ar trebui să fie mici



Folosirea pachetelor împreună cu applet-uri

- Dacă un applet foloseşte un pachet *ne-standard*, atunci *pachetul trebuie să apară în subdirectorul corespunzător* al directorului care conţine fişierul HTML.
  - Acest director poate fi local sau la distanţă — nu contează
  - Variabila de mediu **CLASSPATH** este *ignorată* de applet-uri!



Crearea programelor duale Aplicaţie / Applet

- Dacă o aplicaţie nu are nevoie de operaţii de I/E sau să execute alte sarcini restricţionate, atunci ea poate fi structurată pentru a rula atât ca applet cât şi ca aplicaţie
  - Proiectăm programul ca applet
  - Adăugăm o metodă **main** cu apeluri la **init** şi **start**
  - Adăugăm apeluri la **stop** şi **destroy** în metoda **windowClosing** a rutinei de tratare pentru fereastră
- Un astfel de program poate fi mai versatil

Dualitate Aplicaţie / Applet

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import chapman.io.*;
public class TempConversionApplet extends JApplet {
   // Variabile instanta
   private JLabel l1, l2;       // Etichete
   private JTextField t1, t2;   // Câmpuri text
   private DegCHandler cHnd;    // Tratare ActionEvent
   private DegFHandler fHnd;    // Tratare ActionEvent
   // Metoda de initializare
   public void init() {

      ...
   }
```

metoda **init** din applet

Dualitate Aplicaţie / Applet

```
  // Main method to create frame
  public static void main(String s[]) {
    // Creeaza un cadru care sa contina aplicatia
    JFrame fr = new JFrame("TempConversionApplet ...");
    fr.setSize(250,100);
    // Creeaza+initializeaza un obiect TempConversionApplet
     TempConversionApplet tf = new TempConversionApplet();
     tf.init();
     tf.start();
     // Creeaza un ascultator pentru Window care sa
     // trateze evenimentele "close"
     AppletWindowHandler l = new AppletWindowHandler(tf);
     fr.addWindowListener(l);
     // Adauga obiectul in cetrul cadrului
     fr.getContentPane().add(tf, BorderLayout.CENTER);
     // Afiseaza cadrul
     fr.setVisible( true );
  }
}
```

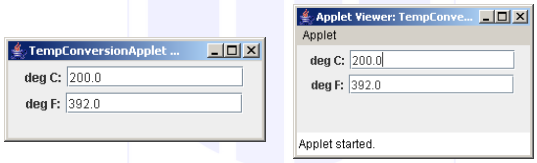
metoda **main** apelează **init** şi **start**

Dualitate Aplicaţie / Applet

```
public class AppletWindowHandler extends WindowAdapter {
   JApplet ap;
   // Constructor
   public AppletWindowHandler ( JApplet a ) { ap = a; }
   // Aceasta metoda implementeaza un ascultator care
   // detecteaza evenimentul "inchidere fereastra",
   // termina ("shut down") applet-ul si opreste programul.
   public void windowClosing(WindowEvent e) {
      ap.stop();
      ap.destroy();
      System.exit(0);
   };
}
```

metoda **windowClosing** apelează **stop** şi **destroy**

Dualitate Aplicaţie / Applet



Rulează ca aplicaţie

Rulează ca applet

Convertirea unei aplicaţii existente la applet (ghid generic)

- Creaţi textul HTML care va invoca applet-ul.
- Clasa cea mai de sus trebuie să extindă **JApplet**, nu **JFrame**. Daţi lui **JApplet** gestionarul **BorderLayout** pe care îl foloseşte implicit **JFrame**.
- Înlocuiţi constructorul clasei cu o metodă **init()** pentru a realiza setările applet-ului.
- Mutaţi orice instrucţiuni de iniţializare din **main()** în **init()**. Alte instrucţiuni care efectuează setări de 're-vizitare a paginii' trebuie plasate într-o metodă numită **start()**.
- Modificaţi/înlăturaţi orice porţiuni de cod care nu pot să fie folosite într-un applet, cum sunt setarea barei de titlu, apelul rutinelor în cod nativ, I/E, comenzi pentru SO etc.
- Plasaţi orice cod pentru curăţare sau oprire în metoda **stop()**.

Conversia Applet -> Aplicaţie

- Faceţi clasa cea mai de sus să extindă **JFrame** şi nu **JApplet**.
- Adăugaţi o rutină **main** clasei. Trebuie avut grijă aici deoarece rutina **main** s-ar putea să trebuiască să transfere argumente în acelaşi fel în care au putut fi transferaţi parametri la applet via HTML.
- Rutina **main** trebuie să creeze o instanţă a clasei. În schimb, constructorul clasei trebuie să apeleze (sau să conţină) metodele **start()** şi **init()**.
- Adăugaţi un meniu cu element de ieşire/buton de ieşire pentru a asigura mijlocul de ieşire din aplicaţie.

Folosirea Swing în applet-uri

- JApplet este un *Container de nivel sus (top level)*
- Unele browsere de Web nu suportă rularea applet-urilor care au componente Swing
- În astfel de cazuri, trebuie instalat *Plug-In* –ul corespunzător
- *Applet Viewer* suportă Swing

```
import javax.swing.*;
public class TestJApplet extends JApplet
{
  public void init()
  {
    // Get the content pane
    Container container = getContentPane();
    JButton button = new JButton("Button");
    container.add(button); // Add it to the content pane
  }
}
```


JApplet în raport cu Applet

- Componentele se adaugă la panoul de conţinut (content pane) al applet-ului Swing, nu direct la applet
- Gestionarii de aranjare se setează la panoul de conţinut al applet-ului Swing, nu direct la applet
- Gestionarul implicit este **BorderLayout** (nu **FlowLayout** ca la applet-urile obişnuite)
- Suportă tehnologiile de asistare (pentru cei cu handicap)
- Suportă adăugarea barelor de meniu

Limitările applet-urilor

- Applet-urile rulează într-o "cutie cu nisip"
- Gestionarul securităţii protejează utilizatorul
- Applet-urile NU pot:
  - Accesa fişiere de pe maşina client.
  - Crea fişiere pe maşina client.
  - Realiza conectări în reţea cu excepţia celor cu maşina de pe care provin.
  - Lansa programe pe maşina client.
  - Încărca biblioteci.
  - Defini apeluri de metode în cod nativ.
  - Opri execuţia interpretorului (nu pot apela `System.exit()`).

Limitările applet-urilor

- Nu pot rula nici un executabil client
- Nu pot comunica cu nici o alta gazdă cu excepţia serverului de unde provin
- Nu pot citi sau scrie pe sistemul de fişierul client
- Pot obţine doar informaţie limitată despre maşina client:
  - Versiunea de Java în uz
  - Numele şi versiunea de SO care rulează
  - Caracterele folosite ca separatori de fişier şi linie
  - Limba folosită pe client (d.e. English) & specificul local (Eastern Europe)
  - Moneda client (€)
- Ferestrele afişate (popped up) de un applet au un mesaj de avertizare

Colecţii Java

Limitările tablourilor

- La gestiunea seriilor, seturilor şi a grupurilor de date, tablourile nu sunt întotdeauna cea mai bună soluţie
- Tablourile nu excelează atunci când datele sunt volatile – în special atunci când mărimea setului de date poate fluctua
  - inserarea unui element necesită glisarea elementelor de deasupra punctului de inserţie -> e nevoie de spaţiu suplimentar alocat la sfârşit
- Mai general, tablourile expun programatorului de aplicaţie problemele legate de gestiunea de nivel jos a memoriei
- Dacă cineva doreşte să ofere accesul la un tablou privat:
  - poate face accesibil tabloul însuşi printr-o metodă accesoare
  - poate furniza o interfaţă pentru iterare cu metodele: first, next, etc.
  - poate întoarce o copie adâncă a tabloului – lucru foarte ineficient

Colecţii versus tablouri

- Lucrul cu API Collections este diferit de lucrul cu tablouri
- Tablourile sunt cu *legare tare la tipuri (strongly typed)*
  - Se specifică tipul elementelor şi compilatorul impune tipul la încercarea de asignare de valori la elemente
  - Se pot defini tablouri de elemente de tipuri primitive
- Colecţiile sunt cu *legare slabă la tipuri (weakly typed)*
  - Există o clasă `Vector` şi toate elementele sale sunt de tipul `Object` -> toate obiectele de toate tipurile pot fi stocate acolo şi e nevoie de forţarea tipului (downcast) elementelor la citire
  - Nu se pot include valori primitive în colecţii, deşi există o cale de împachetare/învelire (box) a lor – clasele învelitoare (d.e. Integer, Boolean, Double etc.)

Colecţii versus tablouri

- Tablourile dau în general viteză mai mare, deoarece reprezintă blocuri de memorie accesibile direct
- Colecţiile sunt obiecte cu metode care trebuie invocate pentru a citi sau scrie elemente
- Colecţiile oferă câteva avantaje generale:
  - Sunt mult mai uşor de folosit pentru programare de uz general, în special atunci când datele sunt foarte volatile – multe adăugări, ştergeri şi modificări directe în timp
  - Designul împărţit "iterator" ajută la ascunderea alegerilor făcute la implementare

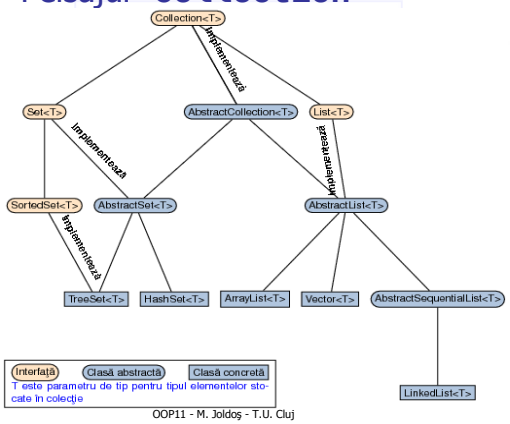
Colecţii

- *Colecţie Java*: orice clasă care păstrează obiecte şi implementează interfaţa **`Collection`**
  - De exemplu, clasa **`ArrayList<T>`** este o clasă colecţie Java şi implementeză toate metodele din interfaţa **`Collection`**
  - Colecţiile sunt folosite împreună cu *iteratori*
- Interfaţa **`Collection`** este cel mai înalt nivel din cadrul general Java pentru clase colecţie
  - Toate clasele colecţie tratate aici se află în pachetul **`java.util`**

Colecţii

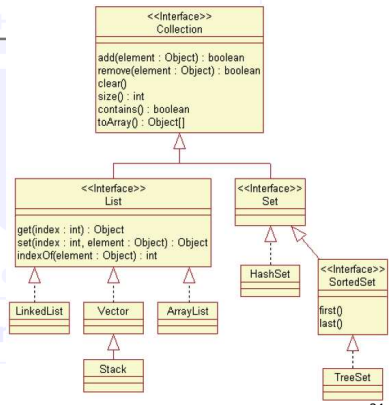
- API Collection include:
  - Colecţii cum sunt **`Vector`**, **`LinkedList`** şi **`Stack`**
  - *Mapări* care indexează valori pe baza cheilor, cum este **`HashMap`**
  - Variante care asigură că elementele sunt întotdeauna ordonate de un comparator: **`TreeSet`** şi **`TreeMap`**
  - *Iteratori* care abstractizează abilitatea de a citi şi scrie conţinutul colecţiilor în bucle şi care izolează acea abilitate de implementarea colecţiei care stă la bază

"Peisajul" **`Collection`**



"Peisajul" **`Collection`**

- Colecţiile ordonate implementează **`List`**
- Colecţiile care asigură unicitatea elementului implementează **`Set`**
- Colecţiile sortate implementează **`SortedSet`**



Caractere de nume nespecificat (wildcards)

- Clasele şi interfeţele din cadrul general Collection pot avea specificări de parametri de tip care nu specifică complet tipul care îl va avea parametrul
  - Pentru că ele specifică o gamă largă de tipuri de argumente, ele sunt numite caractere de nume nespecificat (*wildcards*)

```
public void method(String arg1, ArrayList<?> arg2)
```

  - În exemplul de mai sus, primul argument este de tipul **`String`**, în timp ce al doilea argument poate fi un **`ArrayList<T>`** cu orice tip de bază

Caractere de nume nespecificat (wildcards)

- Se poate limita efectul unui caracter de nume nespecificat (wildcard) precizând că tipul trebuie să fie un tip strămoş sau descendent al unei clase sau a unei interfeţe
  - Notaţia **`<? extends String>`** specifică faptul că argumentul care va fi folosit trebuie să fie un obiect din orice clasă descendentă a lui **`String`**
  - Notaţia **`<? super String>`** specifică faptul că argumentul care va fi folosit trebuie să fie un obiect din orice clasă strămoş al lui **`String`**

Cadrul general `Collection`

- Interfaţa `Collection<T>` descrie operaţiile de bază pe care toate clasele colecţie trebuie să le implementeze
- Cum o interfaţă este un tip, orice metodă poate avea parametri de tipul `Collection<T>`
  - Parametrul respectiv poate fi înlocuit la apel cu orice argument care este un obiect de orice clasă din cadrul general colecţie

Interfaţa `Collection`

- Toate colecţiile pot:
  - adăuga / elimina elemente
  - şterge toate elementele colecţiei astfel încât rezultă un set vid
  - raporta mărimea lor
  - converti datele într-un tablou de `Object`

```
interface Collection
 { // lista partiala de metode
   public int size();
   public void clear();
   public Object[] toArray();
   public boolean add( Object );
   public boolean remove( Object );
   public boolean addAll( Collection );
   public Iterator iterator()
```

- Se pot defini proprietăţi suplimentare definite de implementarea uneia dintre sub-interfeţele `Collection`
  - colecţiile ordonate implementează `List`
  - colecţiile care asigură unicitatea elementelor implementează `Set`
  - colecţiile care sortează implementează `SortedSet`

Construirea colecţiilor

- Colecţiile trebuie *create* explicit
  - Greşeală frecventă: declararea unei referinţe la un `Vector` sau la o `LinkedList` şi presupunerea că obiectul este acolo
- O dată creat obiectul colecţie, pur şi simplu i se adaugă elemente
  - Folosiţi `add` pentru a adăuga un nou element la sfârşit. Atenţie că valorile primitive trebuie "învelite" în clase. E.e.

```
vec.add(new Integer(5))
int i = ((Integer) vec.elementAt(0)).intValue();
Boolean b = ((Boolean)
  vec.elementAt(2)).booleanValue();
```

  - Folosiţi metodele de inserare – definite de subtipuri ale `Collection`
- `remove` (eliminaţi) un element identificând-ul. Multe subtipuri oferă metode de eliminare bazate pe indecşi
- Se poate pune orice obiect Java în orice colecţie
  - colecţii omogene vs eterogene

Capcană: operaţii opţionale

- Atunci când o interfaţă declară o metodă ca "opţională," ea trebuie totuşi implementată într-o clasă care implementează interfaţa
  - Semnificaţia lui "opţional" este că se permite să se scrie o metodă care nu implementează complet interfaţa
  - În orice caz, dacă se dă o implementare trivială atunci corpul metodei trebuie să arunce o excepţie de tipul `UnsupportedOperationException`

Sugestie: Cum să procedăm cu acele excepţii

- Tabelele de metode pentru diversele interfeţe şi clase colecţie indică faptul că se arunca anumite excepţii
  - Acestea sunt excepţii neverificate, aşa că sunt utile la depanare, dar nu e nevoie să fie declarate sau interceptate
- Într-o clasă colecţie existentă, ele pot fi considerate mesaje de eroare la execuţie (run-time error messages)
- Într-o clasă derivată dintr-o altă clasă colecţie, majoritatea sau toate vor fi moştenite
- Într-o clasă colecţie definită din nimic, dacă ea urmează să implementeze o interfaţa colecţie, atunci ea trebuie să arunce excepţiile care sunt specificate în interfaţă

Clase colecţie concrete

- Clasele `ArrayList<T>` şi `Vector<T>` implementează toate metodele din interfaţa `List<T>` şi pot fi folosite aşa cum sunt dacă nu e nevoie de metode suplimentare
  - Fiecare dintre ele se poate folosi atunci când este nevoie de o `List<T>` cu acces aleatoriu eficient la elemente
- Clasa concretă `HashSet<T>` implementează toate metodele din interfaţa `Set<T>` şi poate fi folosită aşa cum este dacă nu e nevoie de metode suplimentare
  - `HashSet<T>` adaugă doar constructori pe lângă metodele din interfaţă
  - `HashSet<T>` este implementată folosind o *tabelă de dispersie*

Clasa `Vector`

- Oferă accesul aleatoriu la o listă scalară de elemente
  - `Vector` şi `ArrayList` au semantica apropiată de un tablou

```
for (int n = 0; n < vec.size(); n++)
   System.out.println((String) vec.elementAt(n));
```

  - Elementele sunt în ordinea în care au fost adăugate la colecţie – nu există sortare implicită
  - Elementele nu trebuie să fie unice în colecţie
- Vectorii se comportă cel mai bine la "acces aleator" la elemente – au în spate tablouri
  - punctul slab – ca şi la tablouri – inserarea şi ştergerea
- Vectorii au capacitate şi mărime
  - `size` = mărimea; numărul de elemente aflate curent în colecţie
  - capacitate = câte locaţii sunt alocate curent pentru elemente;
  - capacitate≥ `size`

Clasa `ArrayList`

- Crearea:
  - `new ArrayList()`
  - `new ArrayList(int initialCapacity)`
- Măsurarea:
  - `int size()`
- Stocarea:
  - `boolean add(Object o)`
  - `boolean add(int index, Object element)`
  - `Object set(int index, Object element)`
- Regăsirea:
  - `Object get(int index)`
  - `Object remove(int index)`
- Testarea:
  - `boolean isEmpty()`
  - `boolean contains(Object elem)`
- Aflarea poziţiei (eşec = -1):
  - `int indexOf(Object elem)`
  - `int lastIndexOf(Object elem)`

Diferenţe între `ArrayList<T>` and `Vector<T>`

- Pentru majoritatea scopurilor, `ArrayList<T>` şi `Vector<T>` sunt echivalente
  - Clasa `Vector<T>` este mai veche şi a trebuit să i se adauge câteva metode pentru a se potrivi în cadrul general colecţie
  - Clasa `ArrayList<T>` este mai noua şi a fost creată ca parte a cadrului general colecţie Java
  - Clasa `ArrayList<T>` se presupune a fi şi mai eficientă decât clasa `Vector<T>`

Exemplu `ArrayList`

```
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorReferenceDemo
{
  public static void main(String[] args)
  {
    ArrayList<Date> birthdays = new
   ArrayList<Date>( );

    birthdays.add(new Date(1, 1, 1990));
    birthdays.add(new Date(2, 2, 1990));
    birthdays.add(new Date(3, 3, 1990));

    System.out.println("Lista contine:");

    Iterator<Date> i = birthdays.iterator( );
    while (i.hasNext( ))
      System.out.println(i.next( ));
    i = birthdays.iterator( );

    Date d = null; //To keep the compiler happy.
      System.out.println("Schimbarea
    referintelor.");
      while (i.hasNext( ))
      {
        d = i.next( );
        d.setDate(4, 1, 1990);
      }

      System.out.println("Lista contine
    acum:");

      i = birthdays.iterator( );
      while (i.hasNext( ))
        System.out.println(i.next( ));

      System.out.println("Pacaleala!");
    }
  }
```


Exemplu: Aflarea şirurilor duplicat

```
import java.util.*;
public class FindDups {
  public static void main(String args[]) {
    Set<String> s = new HashSet<String>();
    for (String a : args)
      if (!s.add(a))
        System.out.println("Duplicat: " + a);
    System.out.println(s.size()+" cuvinte distincte: "+s);
  }
}

java FindDups i came i saw i learned

Duplicat: i
Duplicat: i
4 cuvinte distincte: [i, learned, saw, came]
```

- Remarcaţi faptul că codul referă întotdeauna Colecţia prin tipul interfetei sale (`Set`) nu prin tipul implementarii (`HashSet`).
- Aceasta este o practică de programare *foarte* recomandată deoarece vă oferă flexibilitatea de a schimba implementările prin simpla schimbare a constructorului.

Exemplu modificat : Aflarea şirurilor duplicat

```
import java.util.*;
public class FindDups2 {
  public static void main(String args[]) {
    Set<String> uniques = new HashSet<String>();
    Set<String> dups = new HashSet<String>();
    for (String a : args)
      if (!uniques.add(a)) dups.add(a);

    // Diferenta de multimi distructiva
    uniques.removeAll(dups);
    System.out.println("Cuvinte unice:   " + uniques);
    System.out.println("Cuvinte duplicate: " + dups);
  }
}

java FindDups i came i saw i learned

Cuvinte unice:   [left, saw, came]
Cuvinte duplicate: [i]
```

Clase colecţie concrete

- Clasa concretă **`LinkedList<T>`** este derivată din clasa abstractă **`AbstractSequentialList<T>`**
  - Ar trebui folosită atunci când este nevoie de traversarea secvenţială eficientă a unei liste
- Interfaţa **`SortedSet<T>`** şi clasa concretă **`TreeSet<T>`** sunt destinate să implementeze interfaţa **`Set<T>`** şi să ofere regăsirea rapidă a elementelor
  - Implementarea clasei este asemănătoare cu un arbore binar, dar inserarea păstrează echilibrul arborelui

Clasa **`LinkedList`**

- Este un alt mijloc de obţinere a unei colecţii scalare
  - Fiecare element din lista înlănţuită este discret în memorie
  - Elementul conţine o referinţă spre elementul următor şi alta spre elementul precedent
- Listele înlănţuite se comportă bine la inserări şi ştergeri – nu este nevoie de glisarea elementelor la inserare
  - Se desfac legăturile existente şi se formează altele noi
  - Ştergerea implică schimbarea unor legături
- Iterarea este mai lentă
  - Nu se poate căuta aleator, trebuie traversată element cu element

Capcană: Omiterea lui <T>

- Omiterea lui **`<T>`** sau a numelui de clasă corespunzător dintr-o referinţă la o clasă colecţie este o eroare pentru care compilatorul poate sau nu să emită un mesaj de eroare (depinde de ce conţine codul) şi chiar dacă o face, mesajul de eroare poate fi destul de ciudat
- Căutaţi un **`<T>`** sau un **`<ClassName>`** lipsă atunci când un program care foloseşte clase colecţie dă un mesaj de eroare ciudat sau nu se execută corect

O privire asupra cadrului general Map

- Cadrul general Java *map* tratează colecţii de *perechi ordonate*
  - De exemplu, o cheie şi valoarea asociată ei
- Obiectele din cadrul general map pot implementa funcţii şi relaţii, astfel încât pot fi folosite la construirea claselor pentru baze de date
- Cadrul general map foloseşte interfaţa **`Map<T>`**, clasa **`AbstractMap<T>`** şi clase derivate din clasa **`AbstractMap<T>`**

Enumerări şi iteratori

- Sunt obiecte folosite pentru a parcurge un container.
  - Sunt disponibile pentru unele clase container standard – care implementează interfeţele corespunzătoare.
  - Funcţionează corect chiar dacă containerul se modifică.
  - Ordinea poate sau nu să fie semnificativă.
- Java are doua variaţiuni:
  - **`Enumeration`** (vechi: de la JDK 1.0)
  - **`Iterator`** (mai nou: de la JDK 1.2)

Enumerări

- Pentru a obţine un enumerator **e** pentru containerul **`v`**:
  - **`Enumeration e = v.elements();`**
  - **`e`** este iniţializat la începutul listei.
- Pentru a obţine primul element şi următoarele:
  - **`someObject = e.nextElement()`**
- Pentru a verifica dacă le-am parcurs pe toate:
  - **`e.hasMoreElements()`**
- Exemplu:

```
for (Enumeration e = v.elements();
  e.hasMoreElements(); ) {
    System.out.println(
        e.nextElement());
}
```

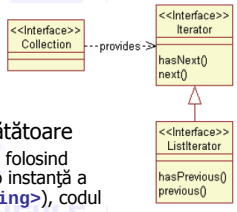
Iteratori

- *Iterator*: un obiect folosit la o colecţie pentru a furniza accesul secvenţial la elementele colecţiei
  - Acest acces permite examinarea şi eventual modificarea elementelor
- Iteratorul impune o ordonare a elementelor colecţiei chiar dacă colecţia în sine nu impune o ordine asupra elementelor pe care le conţine
  - În cazul în care colecţia impune o ordonare asupra elementelor sale, iteratorul va folosi aceeaşi ordonare

Interfaţa `Iterator<T>`

- Interfaţa `Iterator<T>` izolează folosirea unei colecţii de clasa colecţie în sine

```
interface Iterator
{
   public boolean hasNext();
   public Object next();
   public void remove();// optional
}
```



- O `Iterator<T>` nu este de sine stătătoare
  - Ea trebuie asociată cu un obiect colecţie folosind metoda `iterator`. D.e. Dacă `c` este o instanţă a unei clase colecţie (d.e., `HashSet<String>`), codul care urmează obţine un iterator pentru `c`:

```
Iterator iteratorForC = c.iterator();
```


Folosirea unui iterator cu un obiect HashSet<T>

- Un obiect de tipul `HashSet<T>` nu impune nici o ordine asupra elementelor pe care le conţine
- Cu toate acestea, un iterator va impune o ordine asupra elementelor din set
  - Aceasta va fi ordinea în care elementele sunt regăsite de `next()`
  - Deşi la fiecare rulare a programului ordinea elementelor produse astfel poate fi identică, nu există nici o cerinţă care să impună acest lucru

Exemplu: iterator peste `HashSet<T>`

```
import java.util.HashSet;
import java.util.Iterator;
public class HashSetIteratorDemo
{
    public static void main(String[] args)
    {
        HashSet<String> s = new HashSet<String>( );
        s.add("health");
        s.add("love");
        s.add("money");
        System.out.println("The set contains:");
        Iterator<String> i = s.iterator( );
        while (i.hasNext( )) System.out.println(i.next( ));
        i.remove( );
        System.out.println( );
        System.out.println("The set now contains:");
        i = s.iterator( );
        while (i.hasNext( )) System.out.println(i.next( ));
        System.out.println("End of program.");
    }
}
```


Sugestie: Bucle For-Each ca iteratori

- Deşi nu este iterator, bucla for-each poate servi în acelaşi scop ca un iterator
  - Bucla for-each se poate folosi pentru a parcurge fiecare element al unei colecţii
- Buclele for-each pot fi folosite la oricare colecţie menţionată
- Buclele `for` obişnuite nu pot parcurge elementele dintr-un obiect colecţie
  - Spre deosebire de elementele de tablouri, elementele obiectelor colecţie nu sunt în mod normal asociate cu indici
- Deşi bucla `for` obişnuita nu poate parcurge elementele unei colecţii, bucla `for` îmbunătăţită poate parcurge elementele unei colecţii

Bucla "`for each`"

- Sintaxa generală a instrucţiunii `for`-each (pentru fiecare) folosită la o colecţie este

```
for (TipColectie NumeVariabila :
  NumeColectie)
  Instructiune
```

- Linia `for`-each de mai sus trebuie citită ca "pentru fiecare `NumeVariabila` din `NumeColectie`" execută ceea ce urmează.
  - Remarcaţi că `NumeVariabila` trebuie declarată în interiorul fiecărei bucle, nu înainte
  - Remarcaţi, de asemenea, că se foloseşte simbolul "două puncte" (:) după `NumeVariabila`

Exemplu de buclă For-Each ca iterator

```
import java.util.HashSet;
import java.util.Iterator;
public class ForEachDemo {
    public static void main(String[] args) {
        HashSet<String> s = new HashSet<String>( );
        s.add("health");
        s.add("love");
        s.add("money");
        System.out.println("The set contains:");
        String last = null;
        for (String e : s) {
           last = e;
           System.out.println(e);
        }
        s.remove(last);
        System.out.println( );
        System.out.println("The set now contains:");
        for (String e : s) System.out.println(e);
        System.out.println("End of program.");
    }
}
```


Folosirea genericelor

- Un *tip generic* se defineşte în termenii unui alt tip pe care îl colectează sau asupra căruia acţionează în vreun fel, folosind parantezele unghiulare (<>)
  - Exemplu: un **ArrayList<Point>** este un tablou-listă de obiecte **Point** (din pachetul **java.awt**)

```
ArrayList<Point> someList = new ArrayList<Point>();
```

  - permite compilatorului să surprindă o eroare de genul:

```
somelist.add(new Dimension(10, 10));
```

  - Unui obiect colecţie de un anumit tip i se va furniza un iterator specific tipului respectiv

```
Iterator<Point> each = someList.iterator();
```

  - Atunci nu mai este necesar să se forţeze conversia la tipul necesar pentru rezultatele metodelor accesoare, d.e.

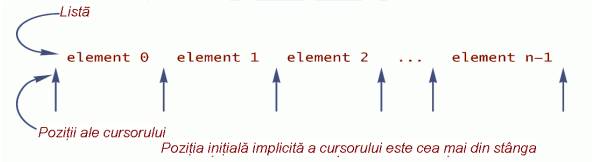
```
while (each.hasNext())
  each.next().x = 11;
```


Interfaţa `ListIterator<T>`

- Interfaţa **`ListIterator<T>`** extinde interfaţa **`Iterator<T>`** şi este menit să lucreze cu colecţii care satisfac interfaţa **`List<T>`**
  - Un **`ListIterator<T>`** are toate metodele pe care le are un **`Iterator<T>`** , plus metode suplimentare
  - Un **`ListIterator<T>`** se poate deplasa în ambele direcţii pe o listă de elemente
  - **`ListIterator<T>`** are metode cum sunt **`set()`** şi **`add()`** care se pot folosi la modificarea elementelor

Cursorul `ListIterator<T>`

- Fiecare **`ListIterator<T>`** are un marcator de poziţie numit *cursor*
  - Dacă lista are *n* elemente, atunci acestea sunt numerotate prin indici de la 0 la *n*-1, dar există *n*+1 poziţii ale cursorului
  - La apelul metodei **`next()`**, se returnează elementul care urmează imediat după cursor, iar cursorul este deplasat înainte cu o poziţie
  - La invocarea metodei **`previous()`** se returnează elementul care urmează imediat înaintea cursorului, iar cursorul este deplasat înapoi cu o poziţie de cursor



Capcană: **`next`** şi **`previous`** pot întoarce o referinţă

- Teoretic, atunci când o operaţie a iteratorului întoarce un element al colecţiei, el poate returna o copie sau o clona a elementului sau poate returna o referinţă la element
- Iteratorii pentru clasele colecţie standard predefinite, cum sunt **`ArrayList<T>`** şi **`HashSet<T>`**, returnează de fapt referinţe
  - De aceea, modificarea valorii returnate va face modificarea elementului din colecţie

Sugestie: Definirea Claselor Iterator proprii

- De obicei nu prea este nevoie de clase **`Iterator<T>`** or **`ListIterator<T>`** definite de programator
- Cea mai simplă şi mai uzitată cale pentru a *defini* o clasă colecţie este *să o facem o clasă derivată* a uneia dintre clasele colecţie de bibliotecă
  - Procedând astfel, metodele **`iterator()`** şi **`listIterator()`** devin automat disponibile programului
- Dacă o clasă colecţie trebuie definită în vreun alt mod, atunci clasa iterator ar trebui definită ca clasă internă (clasă imbricată) în clasa colecţie

Rezumat

- Applet
  - diferenţe faţă de aplicaţiile de sine stătătoare
  - metodele unui applet
  - parametri
  - pachete şi clase folosite de applet
  - aplicaţii duale
  - limitări
  - Swing în applet
- Colecţii Java
  - tablouri vs colecţii
  - API Collections
  - wildcards, generice
  - construirea colecţiilor
  - clase colecţie concrete:
    - ArrayList
    - Vector
    - LinkedList
    - HashSet
    - SortedSet
  - Iteratori
  - Bucla for-each