



## Programare orientată pe obiecte

1. Applet-uri
2. Colecțiile Java



## Ce este un applet Java?

- Applet-urile sunt pur și 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 – aplicațiile să poată fi distribuite automat oricui, pe orice platformă!
- Realitatea – suportul din browsere este neuniform, au apărut limitări impuse de securitate, au apărut alte modalități mai simple de a face același lucru!



## Ce este un applet Java?

- Încă util în situația potrivită
  - client decorat, dezvoltat
  - poate face presupuneri/avea un anumit control asupra tehnologiei client
- De asemenea, foarte util pentru înțelegerea 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 aplicațiilor prin browser și rularea lor independent de browser, folosind JVM din sistemul respectiv.

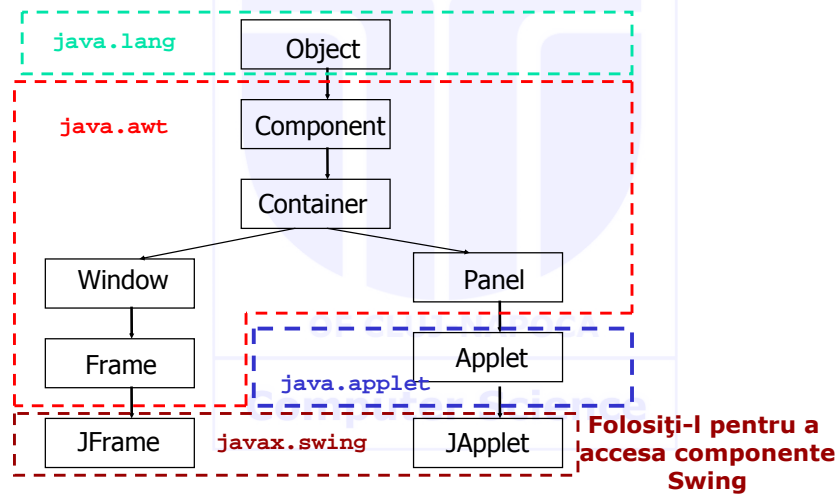


## Utilizarea practică a applet-urilor

- Applet-urile au utilizări serioase pe intranet-uri. Motive:
  - Intranet-urile corporațiilor sunt sigure și administrate -> multe din restricțiile de securitate normale pot fi relaxate.
  - Ușurința administrării. Un număr mare de aplicații foarte mici distribuite peste foarte multe mașini poate crea o problemă de administrare (d.e. actualizarea periodică) – utilizatorii mai puțin capabili le pot muta, deteriora sau șterge. Applet-urile de pe un server central, încărcate la cerere, nu au această problemă (deși este nevoie să se întrețină browserele, plug-in-urile etc.).
  - Lărgimea de bandă mare. Rețelele 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 aplicații care au nevoie de o interfață mai robustă decât aceea dintr-o pagină HTML cu JavaScript.



## Arborele de descendență pentru Applet



OOP11 - M. Joldoș - T.U. Cluj

5



## Diferențe esențiale între applet-uri și aplicațiile de sine stătătoare

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

OOP11 - M. Joldoș - T.U. Cluj

6



## 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 animațiile care se execută atunci când applet-ul este (din nou) vizibil
  - `stop()` — apelată de browser pentru a opri animațiile în execuție 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

OOP11 - M. Joldoș - T.U. Cluj

7



## Metodele unui applet

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

OOP11 - M. Joldoș - T.U. Cluj

8



## 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 și le adăugăm la *panoul de conținut* (*content pane*) al containerului `JApplet`.
  - Creăm obiecte "ascultător" pentru a detecta și a răspunde la evenimentele așteptate de fiecare componentă GUI și asignăm ascultătorii componentelor corespunzătoare.
  - Creăm un fișier text HTML pentru a preciza browser-ului care applet Java trebuie încărcat și 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` și `moșteneste` toate celelalte metode fără efect

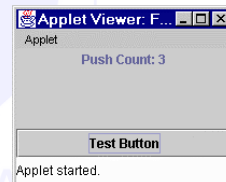
Remarcați că toate componentele sunt adăugate la `ContentPane` al `JApplet`



## 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 stocați ca parte a paginii de Web care conține applet-ul.
- Parametrii sunt creați folosind eticheta HTML `<PARAM>` și cele două atribute ale sale: `NAME` și `VALUE`.
- Pot exista mai multe etichete `<PARAM>` pentru un applet. Toate trebuie incluse între eticheta de început `<APPLET>` și cea de sfârșit `</APPLET>`. Exemplu cu mai mulți 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 folosește pentru a da un nume parametrului. Atributul `VALUE` dă parametrului numit o valoare.
- Obținerea parametrilor: metoda `getParameter`. De.e. `String display1 = getParameter("Headline1");`



## Etichete HTML pentru applet-uri

```

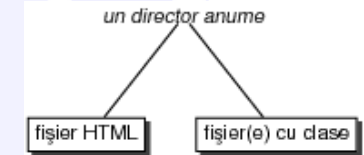
<APPLET
[CODEBASE = URLbazaPentruCod]
CODE = fisierApplet
[ATL = textAlternativ]
[NAME = numeInstantaApplet]
WIDTH = latimeInPixeli HEIGHT = inaltimeInPixeli
[ALIGN = alignment]
[VSPACE = spatiuVertInPixeli] [HSPACE =
spatiuOrizInPixeli]
[<PARAM NAME = numeParametru1 VALUE = valoareParametru1>]
[<PARAM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

```



## Locul în care se află fișierele clasă pentru applet-uri

- Dacă un applet folosește o clasă care *nu* se află într-un pachet, clasa respectivă trebuie să fie prezentă în *același director* ca și fișierul HTML folosit pentru lansarea applet-ului

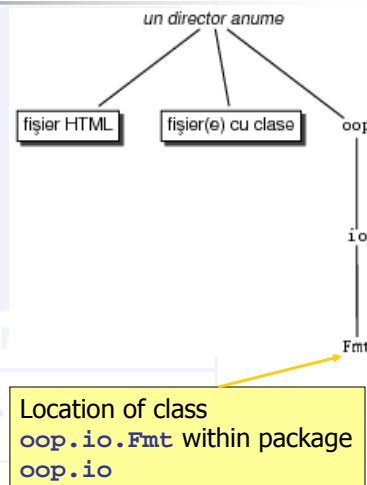


- Acest director poate fi local sau la distanță — nu contează
- Fișierele clasă pot fi transferate peste rețea, așa că ar trebui să fie mici



## Folosirea pachetelor împreună cu applet-uri

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



## Crearea programelor duale Aplicație / Applet

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



## Dualitate Aplicație / 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 Aplicație / 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 centrul cadrului
    fr.getContentPane().add(tf, BorderLayout.CENTER);
    // Afiseaza cadrul
    fr.setVisible(true);
}
}
```

metoda main apelează init și start



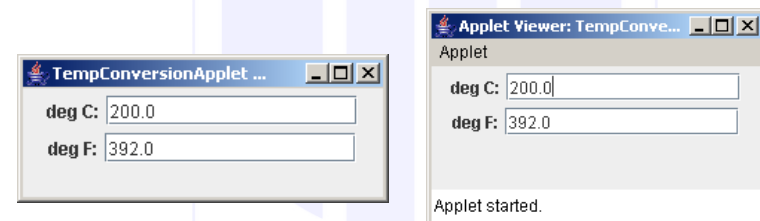
## Dualitate Aplicație / Applet

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

metoda windowClosing apelează stop și destroy



## Dualitate Aplicație / Applet



Rulează ca aplicație

Rulează ca applet



## Convertirea unei aplicații existente la applet (ghid generic)

- Creați textul HTML care va invoca applet-ul.
- Clasa cea mai de sus trebuie să extindă **JApplet**, nu **JFrame**. Dați lui **JApplet** gestionarul **BorderLayout** pe care îl folosește implicit **JFrame**.
- Înlocuiți constructorul clasei cu o metodă **init()** pentru a realiza setările applet-ului.
- Mutați orice instrucțiuni de inițializare din **main()** în **init()**. Alte instrucțiuni care efectuează setări de 're-visitare a paginii' trebuie plasate într-o metodă numită **start()**.
- Modificați/înlăturați orice porțiuni 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.
- Plasați orice cod pentru curățare sau oprire în metoda **stop()**.



## Conversia Applet -> Aplicație

- Faceți clasa cea mai de sus să extindă **JFrame** și nu **JApplet**.
- Adăugați o rutină **main** clasei. Trebuie avut grijă aici deoarece rutina **main** s-ar putea să trebuiască să transfere argumente în același fel în care au putut fi transferați parametri la applet via HTML.
- Rutina **main** trebuie să creeze o instanță a clasei. În schimb, constructorul clasei trebuie să apeleze (sau să conțină) metodele **start()** și **init()**.
- Adăugați un meniu cu element de ieșire/buton de ieșire pentru a asigura mijlocul de ieșire din aplicație.



## 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 conținut (content pane) al applet-ului Swing, nu direct la applet
- Gestionarii de aranjare se setează la panoul de conținut al applet-ului Swing, nu direct la applet
- Gestionarul implicit este **BorderLayout** (nu **FlowLayout** ca la applet-urile obișnuite)
- 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 securității protejează utilizatorul
- Applet-urile NU pot:
  - Accesa fișiere de pe mașina client.
  - Crea fișiere pe mașina client.
  - Realiza conectări în rețea cu excepția celor cu mașina de pe care provin.
  - Lansa programe pe mașina client.
  - Încărca biblioteci.
  - Defini apeluri de metode în cod nativ.
  - Opri execuția 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 excepția serverului de unde provin
- Nu pot citi sau scrie pe sistemul de fișierul client
- Pot obține doar informație limitată despre mașina client:
  - Versiunea de Java în uz
  - Numele și versiunea de SO care rulează
  - Caracterele folosite ca separatori de fișier și linie
  - Limba folosită pe client (d.e. English) & specificul local (Eastern Europe)
  - Moneda client (€)
- Ferestrele afișate (popped up) de un applet au un mesaj de avertizare



## Colecții Java



## Limitările tablourilor

- La gestiunea seriilor, seturilor și a grupurilor de date, tablourile nu sunt întotdeauna cea mai bună soluție
- 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 inserție -> e nevoie de spațiu suplimentar alocat la sfârșit
- Mai general, tablourile expun programatorului de aplicație problemele legate de gestiunea de nivel jos a memoriei
- Dacă cineva dorește să ofere accesul la un tablou privat:
  - poate face accesibil tabloul însuși printr-o metodă accesoare
  - poate furniza o interfață pentru iterare cu metodele: first, next, etc.
  - poate întoarce o copie adâncă a tabloului – lucru foarte ineficient



## Colecții 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 și compilatorul impune tipul la încercarea de asignare de valori la elemente
  - Se pot defini tablouri de elemente de tipuri primitive
- Colecțiile sunt cu *legare slabă la tipuri (weakly typed)*
  - Există o clasă **Vector** și toate elementele sale sunt de tipul **Object** -> toate obiectele de toate tipurile pot fi stocate acolo și e nevoie de forțarea tipului (downcast) elementelor la citire
  - Nu se pot include valori primitive în colecții, deși există o cale de împachetare/invelire (box) a lor – clasele învelitoare (d.e. Integer, Boolean, Double etc.)



## Colecții versus tablouri

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



## Colecții

- **Colecție Java**: orice clasă care păstrează obiecte și implementează interfața **Collection**
  - De exemplu, clasa **ArrayList<T>** este o clasă colecție Java și implementează toate metodele din interfața **Collection**
  - Colecțiile sunt folosite împreună cu *iteratori*
- Interfața **Collection** este cel mai înalt nivel din cadrul general Java pentru clase colecție
  - Toate clasele colecție tratate aici se află în pachetul **java.util**



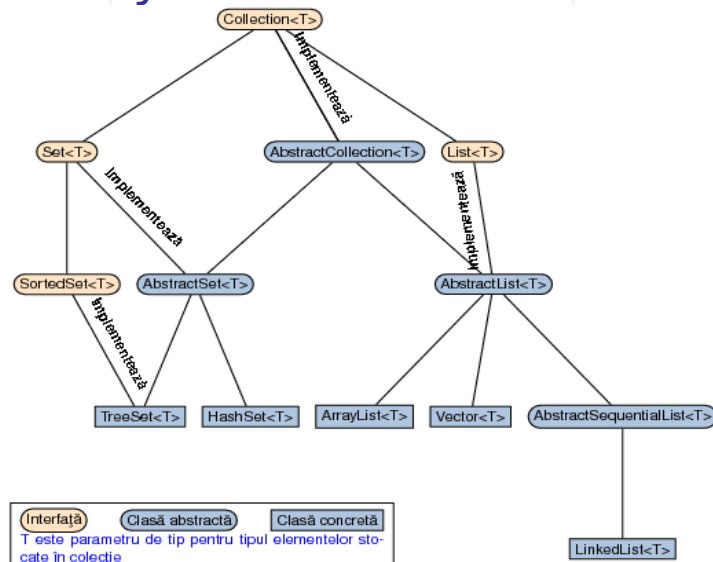
## Colecții

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





## "Peisajul" Collection



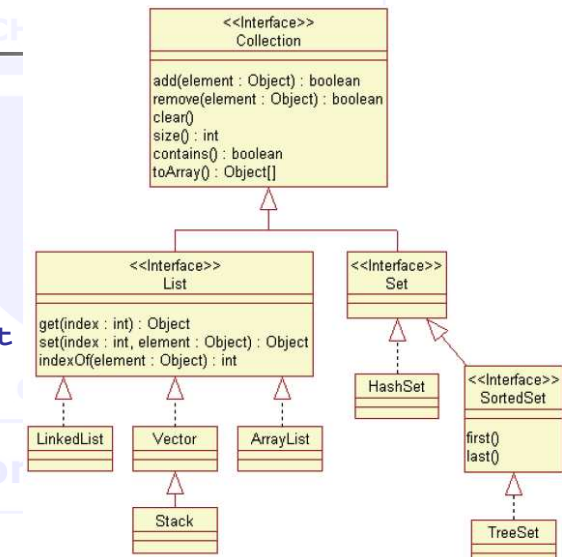
OOP11 - M. Joldoș - T.U. Cluj

33



## "Peisajul" Collection

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



OOP11 - M. Joldoș - T.U. Cluj

34



## Caractere de nume nespecificat (wildcards)

- Clasele și interfețele 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

OOP11 - M. Joldoș - T.U. Cluj

35



## Caractere de nume nespecificat (wildcards)

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

OOP11 - M. Joldoș - T.U. Cluj

36



## Cadrul general Collection

- Interfața `Collection<T>` descrie operațiile de bază pe care toate clasele colecție trebuie să le implementeze
- Cum o interfață 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 colecție



## Interfața Collection

- Toate colecțiile pot:
    - adăuga / elimina elemente
    - șterge toate elementele colecției astfel încât rezultă un set vid
    - raporta mărimea lor
    - converti datele într-un tablou de `Object`
  - Se pot defini proprietăți suplimentare definite de implementarea uneia dintre sub-interfețele `Collection`
    - colecțiile ordonate implementează `List`
    - colecțiile care asigură unicitatea elementelor implementează `Set`
    - colecțiile care sortează implementează `SortedSet`
- ```
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()
```



## Construirea colecțiilor

- Colecțiile trebuie *create* explicit
  - Greșeală frecventă: declararea unei referințe la un `Vector` sau la o `LinkedList` și presupunerea că obiectul este acolo
- O dată creat obiectul colecție, pur și simplu i se adaugă elemente
  - Folosiți `add` pentru a adăuga un nou element la sfârșit. Atenție 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();
```
  - Folosiți metodele de inserare – definite de subtipuri ale `Collection`
- `remove` (eliminați) un element identificându-l. Multe subtipuri oferă metode de eliminare bazate pe indecși
- Se poate pune orice obiect Java în orice colecție
  - colecții omogene vs eterogene



## Capcană: operații optionale

- Atunci când o interfață declară o metodă ca "opțională," ea trebuie totuși implementată într-o clasă care implementează interfața
  - Semnificația lui "opțional" este că se permite să se scrie o metodă care nu implementează complet interfața
  - În orice caz, dacă se dă o implementare trivială atunci corpul metodei trebuie să arunce o excepție de tipul `UnsupportedOperationException`



## Sugestie: Cum să procedăm cu acele excepții

- Tabelele de metode pentru diversele interfețe și clase colecție indică faptul că se arunca anumite excepții
  - Acestea sunt excepții neverificate, așa că sunt utile la depanare, dar nu e nevoie să fie declarate sau interceptate
- Într-o clasă colecție existentă, ele pot fi considerate mesaje de eroare la execuție (run-time error messages)
- Într-o clasă derivată dintr-o altă clasă colecție, majoritatea sau toate vor fi moștenite
- Într-o clasă colecție definită din nimic, dacă ea urmează să implementeze o interfața colecție, atunci ea trebuie să arunce excepțiile care sunt specificate în interfață



## Clase colecție concrete

- Clasele `ArrayList<T>` și `Vector<T>` implementează toate metodele din interfața `List<T>` și pot fi folosite așa 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 interfața `Set<T>` și poate fi folosită așa cum este dacă nu e nevoie de metode suplimentare
  - `HashSet<T>` adaugă doar constructori pe lângă metodele din interfață
  - `HashSet<T>` este implementată folosind o *tabelă de dispersie*



## Clasa Vector

- Oferă accesul aleatoriu la o listă scalară de elemente
  - `Vector` și `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 colecție – nu există sortare implicită
  - Elementele nu trebuie să fie unice în colecție
- Vectorii se comportă cel mai bine la "acces aleator" la elemente – au în spate tablouri
  - punctul slab – ca și la tablouri – inserarea și ștergerea
- Vectorii au capacitate și mărime
  - `size` = mărimea; numărul de elemente aflate curent în colecție
  - capacitate = câte locații 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 poziției (eșec = -1):
  - `int indexOf(Object elem)`
  - `int lastIndexOf(Object elem)`



## Diferențe între ArrayList<T> and Vector<T>

- Pentru majoritatea scopurilor, **ArrayList<T>** și **Vector<T>** sunt echivalente
  - Clasa **Vector<T>** este mai veche și a trebuit să i se adauge câteva metode pentru a se potrivi în cadrul general colecție
  - Clasa **ArrayList<T>** este mai nouă și a fost creată ca parte a cadrului general colecție Java
  - Clasa **ArrayList<T>** se presupune a fi și 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 șirurilor duplicat

```
import java.util.*;
public class FindDups {
    public static void main(String args[]) {
        Set<String> s = new HashSet<String>(0);
        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]

- Remarcați faptul că **codul referă întotdeauna Colecția prin tipul interfeței sale (Set)** nu prin tipul implementării (**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 șirurilor duplicat

```
import java.util.*;
public class FindDups2 {
    public static void main(String args[]) {
        Set<String> uniques = new HashSet<String>(0);
        Set<String> dups = new HashSet<String>(0);
        for (String a : args)
            if (!uniques.add(a)) dups.add(a);

        // Diferența de mulțimi distructivă
        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 colecție concrete

- Clasa concretă `LinkedList<T>` este derivată din clasa abstractă `AbstractSequentialList<T>`
  - Ar trebui folosită atunci când este nevoie de traversarea secvențială eficientă a unei liste
- Interfața `SortedSet<T>` și clasa concretă `TreeSet<T>` sunt destinate să implementeze interfața `Set<T>` și 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 obținere a unei colecții scalare
  - Fiecare element din lista înlănțuită este discret în memorie
  - Elementul conține o referință spre elementul următor și alta spre elementul precedent
- Listele înlănțuite se comportă bine la inserări și ștergeri – nu este nevoie de glisarea elementelor la inserare
  - Se desfac legăturile existente și se formează altele noi
  - Ștergerea 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 referință la o clasă colecție este o eroare pentru care compilatorul poate sau nu să emită un mesaj de eroare (depinde de ce conține codul) și chiar dacă o face, mesajul de eroare poate fi destul de ciudat
- Căutați un `<T>` sau un `<ClassName>` lipsă atunci când un program care folosește clase colecție dă un mesaj de eroare ciudat sau nu se execută corect



## O privire asupra cadrului general `Map`

- Cadrul general Java *map* tratează colecții de *perechi ordonate*
  - De exemplu, o cheie și valoarea asociată ei
- Obiectele din cadrul general `map` pot implementa funcții și relații, astfel încât pot fi folosite la construirea claselor pentru baze de date
- Cadrul general `map` folosește interfața `Map<T>`, clasa `AbstractMap<T>` și clase derivate din clasa `AbstractMap<T>`





## Enumerări și iteratori

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



## Enumerări

- Pentru a obține un enumerator `e` pentru containerul `v`:
  - `Enumeration e = v.elements();`
  - `e` este inițializat la începutul listei.
- Pentru a obține primul element și 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

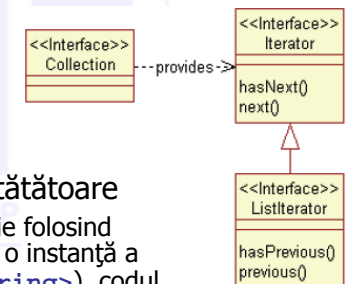
- **Iterator**: un obiect folosit la o colecție pentru a furniza accesul secvențial la elementele colecției
  - Acest acces permite examinarea și eventual modificarea elementelor
- Iteratorul impune o ordonare a elementelor colecției chiar dacă colecția în sine nu impune o ordine asupra elementelor pe care le conține
  - În cazul în care colecția impune o ordonare asupra elementelor sale, iteratorul va folosi aceeași ordonare



## Interfața `Iterator<T>`

- Interfața `Iterator<T>` izolează folosirea unei colecții de la clasa colecției 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 colecție folosind metoda `iterator`. D.e. Dacă `c` este o instanță a unei clase colecție (d.e., `HashSet<String>`), codul care urmează obține 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 conține
- Cu toate acestea, un iterator va impune o ordine asupra elementelor din set
  - Aceasta va fi ordinea în care elementele sunt regăsite de `next()`
  - Deși la fiecare rulare a programului ordinea elementelor produse astfel poate fi identică, nu există nici o cerință 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

- Deși nu este iterator, bucla for-each poate servi în același scop ca un iterator
  - Bucla for-each se poate folosi pentru a parcurge fiecare element al unei colecții
- Buclele for-each pot fi folosite la oricare colecție menționată
- Buclele `for` obișnuite nu pot parcurge elementele dintr-un obiect colecție
  - Spre deosebire de elementele de tablouri, elementele obiectelor colecție nu sunt în mod normal asociate cu indici
- Deși bucla `for` obișnuită nu poate parcurge elementele unei colecții, bucla `for` îmbunătățită poate parcurge elementele unei colecții



## Bucle "for each"

- Sintaxa generală a instrucțiunii `for-each` (pentru fiecare) folosită la o colecție este
 

```
for (TipColecție NumeVariabila :
    NumeColecție)
    Instructiune
```
- Linia `for-each` de mai sus trebuie citită ca "pentru fiecare `NumeVariabila` din `NumeColecție`" execută ceea ce urmează.
  - Remarcați că `NumeVariabila` trebuie declarată în interiorul fiecărei bucle, nu înainte
  - Remarcați, de asemenea, că se folosește 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.");
    }
}
```

OOP11 - M. Joldoș - T.U. Cluj

61



## Folosirea genericelor

- Un *tip generic* se definește în termenii unui alt tip pe care îl colectează sau asupra căruia acționează 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 colecție 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 forțeze conversia la tipul necesar pentru rezultatele metodelor accesor, d.e.
- ```
while (each.hasNext())
    each.next().x = 11;
```

OOP11 - M. Joldoș - T.U. Cluj

62



## Interfața ListIterator<T>

- Interfața **ListIterator<T>** extinde interfața **Iterator<T>** și este menită să lucreze cu colecții care satisfac interfața **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 direcții pe o listă de elemente
- ListIterator<T>** are metode cum sunt **set()** și **add()** care se pot folosi la modificarea elementelor

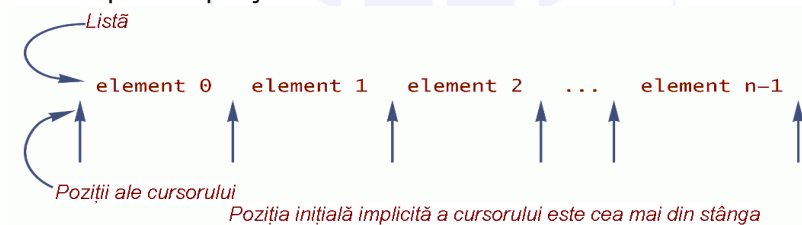
OOP11 - M. Joldoș - T.U. Cluj

63



## Cursorul ListIterator<T>

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



OOP11 - M. Joldoș - T.U. Cluj

64



## Capcană: `next` și `previous` pot întoarce o referință

- Teoretic, atunci când o operație a iteratorului întoarce un element al colecției, el poate returna o copie sau o clona a elementului sau poate returna o referință la element
- Iteratorii pentru clasele colecție standard predefinite, cum sunt `ArrayList<T>` și `HashSet<T>`, returnează de fapt referințe
  - De aceea, modificarea valorii returnate va face modificarea elementului din colecție



## Sugestie: Definirea Claselor Iterator proprii

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



## Rezumat

- Applet
  - diferențe față de aplicațiile de sine stătătoare
  - metodele unui applet
  - parametri
  - pachete și clase folosite de applet
  - aplicații duale
  - limitări
  - Swing în applet
- Colecții Java
  - tablouri vs colecții
  - API Collections
  - wildcards, generice
  - construirea colecțiilor
  - clase colecție concrete:
    - `ArrayList`
    - `Vector`
    - `LinkedList`
    - `HashSet`
    - `SortedSet`
  - Iteratori
  - Bucla for-each