Programare orientată pe obiecte

1. Colecţiile Java

Limitările tablourilor

- Tablourile nu sunt întotdeauna cea mai bună soluţie pentru gestiunea seriilor, seturilor şi a grupurilor de date
- Tablourile nu excelează 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
- 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 colecții (*API Collection*) 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 de execuție mai mare, deoarece reprezintă blocuri de memorie accesibile direct
- Colecţiile sunt obiecte cu metode care trebuie invocate pentru a citi sau scrie elemente
- Colecţiile 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

Colecţii

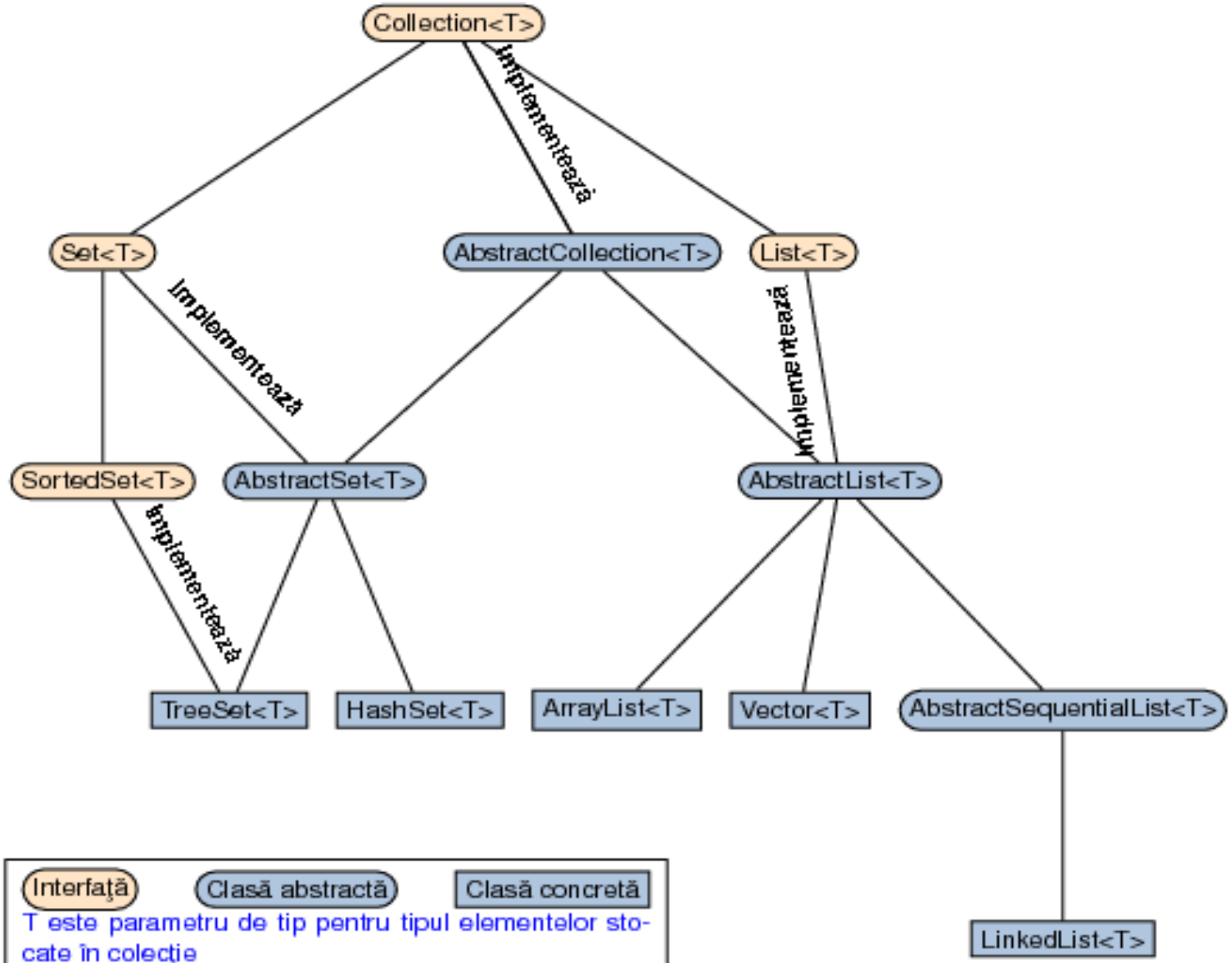
- *Colecţie Java*: orice clasă care păstrează obiecte şi implementează interfaţa **`Collection`**
  - De exemplu, clasa **`ArrayList<T>`** este o clasă colecţie Java şi implementeză toate metodele din interfaţa **`Collection`**
  - Colecţiile sunt folosite împreună cu *iteratori*

- 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

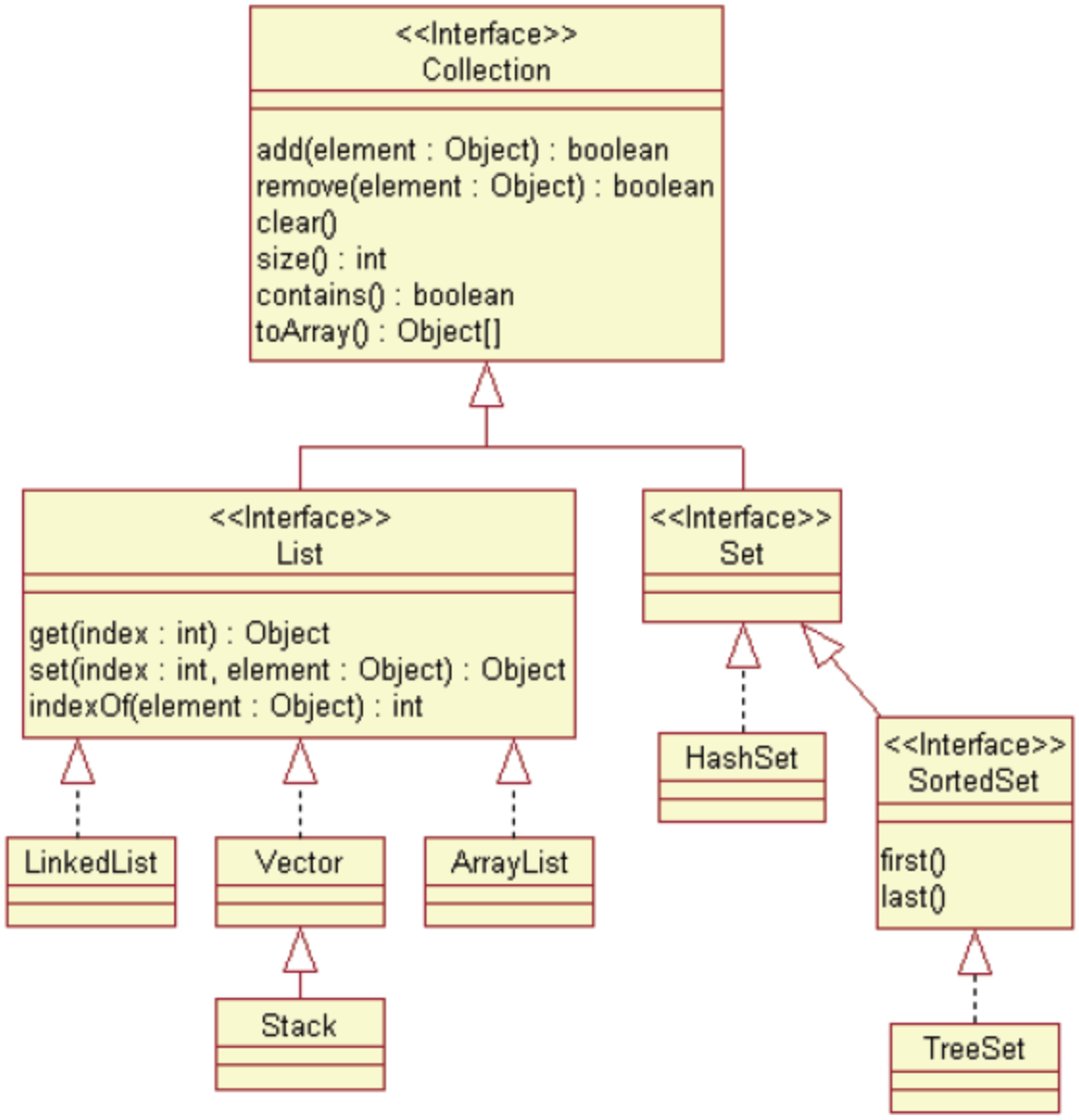
- *API Collection* include:
  - Colecţii cum sunt **`Vector`**, **`LinkedList`** şi **`Stack`**
  - *Mapări* care indexează valori pe baza cheilor, cum este **`HashMap`**
  - Variante care asigură că elementele sunt întotdeauna ordonate de un comparator: **`TreeSet`** şi **`TreeMap`**
  - *Iteratori* 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:

```
Vector<Integer> vec = new Vector<Integer>();
vec.add(new Integer(5));
vec.add(new Integer(4));
System.out.println(vec); // Output: [5, 4]
int i = ((Integer) vec.elementAt(0)).intValue();
```

  - Folosiţi metodele de inserare – definite de subtipuri ale `Collection`
- Folosiți `remove` pentru a elimina un element, identificându-l
  - Multe subtipuri oferă metode de eliminare bazate pe indecşi
- Se poate pune orice obiect Java în orice colecţie
  - Colecţii omogene vs eterogene

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 = este dimensiunea tabloului `Object[] elementData` din clasa Vector; capacitatea se incrementează automat cu `capacityIncrement` de fiecare dată cand `size` devine mai mare decat capacitatea
  - 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 nouă 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;
import java.util.Date;

public class IteratorReferenceDemo
{
  public static void main(String[] args)
  {
    ArrayList<Date> birthdays =
                  new ArrayList<Date>( );
    birthdays.add(new Date(90, 1, 1));
    birthdays.add(new Date(90, 2, 2));
    birthdays.add(new Date(90, 3, 3));
    System.out.println("Lista contine:");
    Iterator<Date> i =
   birthdays.iterator();
    while (i.hasNext( ))
       System.out.println(i.next( ));
   i = birthdays.iterator( );
   Date d = null;
   System.out.println("Schimbarea
                       referintelor.");
   while (i.hasNext( ))  {
      d = i.next( );
      d.setDate(1);
      d.setMonth(3);
      d.setYear(90);
   }
   System.out.println("Lista contine
    acum:");
   i = birthdays.iterator( );
   while (i.hasNext( ))
      System.out.println(i.next( ));
   }
}
```

Rezultate afisate:

```
Lista contine:
Thu Feb 01 00:00:00 EET 1990
Fri Mar 02 00:00:00 EET 1990
Tue Apr 03 00:00:00 EEST 1990
Schimbarea referintelor.
Lista contine acum:
Sun Apr 01 00:00:00 EEST 1990
Sun Apr 01 00:00:00 EEST 1990
Sun Apr 01 00:00:00 EEST 1990
```

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);
   }
}
```

Rulare:

```
java FindDups i came i saw i learned
```

Rezultate afişate:

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

- Remarcaţi faptul că codul referă întotdeauna Colecţia 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 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);
   }
}
```

Rulare:

```
java FindDups i came i saw i learned
```

Rezultate afişate:

```
Cuvinte unice:    [learned, 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

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 aceasta

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 două 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());
  }
```

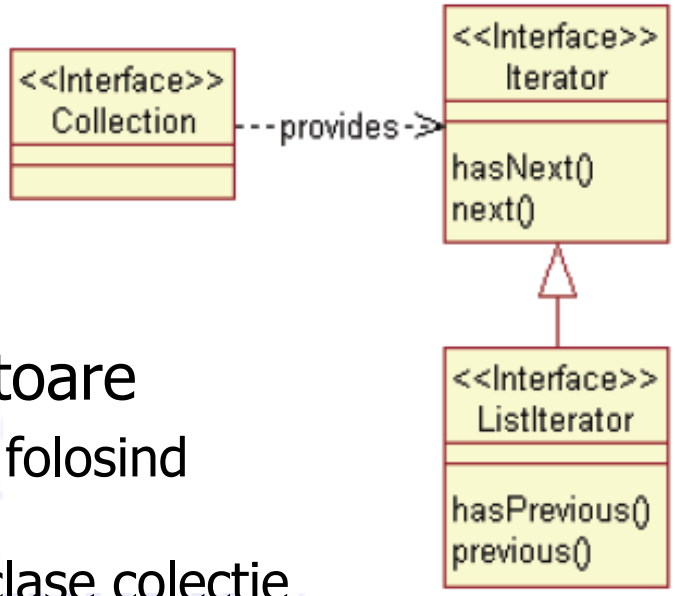
Iteratori

- *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
}
```



- `Iterator<T>` nu este de sine stătătoare
  - Ea trebuie asociată cu un obiect colecţie folosind metoda `iterator`
  - Exemplu: 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.");
 }
}
```

Rezultate afișate:

```
The set contains:
love
money
health

The set now contains:
love
money
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 din tablouri, elementele obiectelor colecţie nu sunt în mod normal asociate cu indici
- Deşi bucla `for` obişnuită 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.");
  }
}
```

Rezultate afișate:

```
The set contains:
love
money
health

The set now contains:
love
money
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.

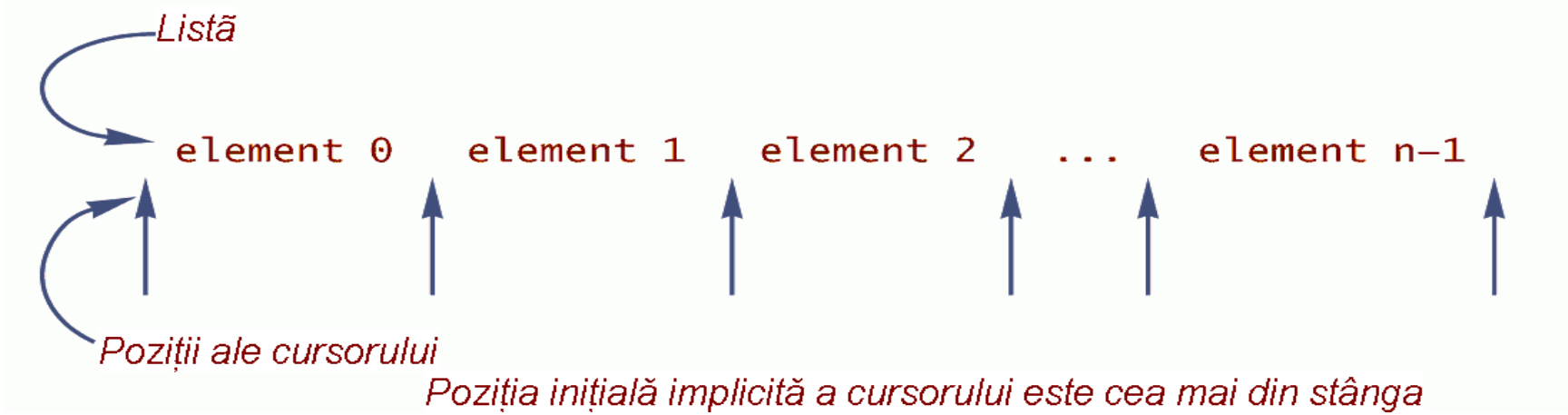
```
while (each.hasNext())
   each.next().x = 11;
```

Interfaţa `ListIterator<T>`

- Interfaţa `ListIterator<T>` extinde interfaţa `Iterator<T>` şi este menită 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 clonă 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>` sau `ListIterator<T>` definite de programator

- Cea mai simplă şi mai utilizată 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

Sortarea colecțiilor/tablourilor

- Folosind metoda statică **sort** a clasei **Collections** (colecții)
- Folosind metoda statică **sort** a clasei **Arrays** (tablouri)
- Sortarea unei colectii/tablou după criteriul definit de metoda ***compareTo*** a clasei din care fac parte obiectele
- Exemplu (sortare colecție):

```
import java.util.*;
class TestSort {
  public static void main(String[] args) {
    ArrayList<String> stuff = new ArrayList<String>(); // #1
    stuff.add("Denver");
    stuff.add("Boulder");
    stuff.add("Vail");
    stuff.add("Aspen");
    stuff.add("Telluride");
    System.out.println("unsorted " + stuff);
    Collections.sort(stuff); // #2
    System.out.println("sorted " + stuff);
  }
}
```

Rezultate afișate:

```
unsorted [Denver, Boulder, Vail, Aspen, Telluride]
sorted [Aspen, Boulder, Denver, Telluride, Vail]
```

Sortarea colecțiilor/tablourilor

- Definirea mai multor criterii de sortare ale aceleiași colecții/aceluiași tablou folosind **comparatori**
  - Comparatorii sunt folosiți ca și argumente la ceva ce sortează
    - Metode de sortare
    - Structuri de date care sortează
  - Se utilizează interfața **java.util.Comparator**
    - Sunt create obiecte care sunt transmise metodelor de sortare sau structurilor de date care sortează
    - Un Comparator trebuie să definească o metodă ***compare*** care primește ca și argumente două Object și returnează un întreg <0, 0, sau >0 (mai mic, egal, mai mare) comparând primul Object cu cel de-al doilea

Sortarea colecțiilor/tablourilor

- Exemplu
  - Listarea conținutului directorului implicit (home) al unui utilizator
  - Demonstrează folosirea interfeței Comparator pentru a sorta același tablou după două criterii diferite
    - Directoarele înaintea fișierelor, apoi alfabetic
    - După lungimea numelui de fișier/director, cel mai lung primul

Sortarea colecțiilor/tablourilor

```
// Author: Fred Swartz 2006-Aug-23  Public domain.

import java.util.Arrays;
import java.util.Comparator;
import java.io.*;

public class Filelistsort {

    //================================================== main
    public static void main(String[] args) {
        //... Creaza comparatorii pentru sortare.
        Comparator<File> byDirThenAlpha = new DirAlphaComparator();
        Comparator<File> byNameLength   = new NameLengthComparator();

        //... Creaza un obiect a File pentru directorul utilizatorului.
        File dir = new File(System.getProperty("user.home"));
        File[] children = dir.listFiles();
```

Sortarea colecțiilor/tablourilor

```
        System.out.println("Fisierele dupa director, apoi alfabetic ");
        Arrays.sort(children, byDirThenAlpha);
        printFileNames(children);

        System.out.println("Fisierele dupa lungimea numelui lor
                                              (cel mai lung primul)");
        Arrays.sort(children, byNameLength);
        printFileNames(children);
    }

    //========================================== printFileNames
    private static void printFileNames(File[] fa){
        for (File oneEntry : fa) {
            System.out.println("   " + oneEntry.getName());
        }
    }
}
```

Sortarea colecțiilor/tablourilor

```
////////////////////////////////////////////////// DirAlphaComparator
// Pentru a sorta directoarele inaintea fisierelor, apoi alfabetic.
class DirAlphaComparator implements Comparator<File> {

    // Interfata Comparator necesita definirea metodei compare.
    public int compare(File filea, File fileb) {
        //... Sorteaza directoarele inaintea fisierelor,
        //    altfel alfabetic fara a tine seama de majuscule/minuscule.
        if (filea.isDirectory() && !fileb.isDirectory()) {
            return -1;

        } else if (!filea.isDirectory() && fileb.isDirectory()) {
            return 1;

        } else {
            return filea.getName().compareToIgnoreCase(fileb.getName());
        }
    }
}
```

Sortarea colecțiilor/tablourilor

```
/////////////////////////////////////////////////// NameLengthComparator
// Pentru a sorta dupa lungimea numelui de fisier/director
// (cel mai lung primul).
class NameLengthComparator implements Comparator<File> {

    // Interfata Comparator necesita definirea metodei compare.
    public int compare(File filea, File fileb) {
        int comp = fileb.getName().length() - filea.getName().length();
        if (comp != 0) {
            //... daca lungimile sunt diferite, am terminat.
            return comp;
        } else {
            //... daca lungimile sunt egale, sorteaza alfabetic.
            return filea.getName().compareToIgnoreCase(fileb.getName());
        }
    }
}
```