PROGRAMARE ȘI STRUCTURI DE DATE

CURS 4

Lect. dr. Oneț - Marian Zsuzsanna

Facultatea de Matematică și Informatică UBB
în colaborare cu NTT Data

Cuprins

Containere I

- Un *container* este o grupare de date în care se pot adăuga (insera) elemente noi și din care se pot șterge elemente.
- Un container în general ne permite următoarele operații:
  - adăugarea unui element
  - ștergerea unui element
  - returnarea numărului de elemente din container
  - accesarea elementelor (în general folosind *iteratorul*)
  - căutarea unui element
- Lista operațiilor pentru un container (împreună cu precondiții și postcondiții) reprezintă *interfața* containerului.

Containere II

- Poate cunoașteți deja containerul *Lista* (sau *ArrayList*) - un container în care fiecare element are o poziție bine stabilită -, dar pot exista și alte containere, în funție de diferite criterii:
  - Dacă elementele se pot repeta sau nu
  - Dacă elemente au o poziție sau nu
  - Dacă elementele stocate sunt elemente simple sau perechi
  - Dacă orice element poate fi accesat din container sau doar anumite elemente

Containere III

- Pentru a lucra cât mai general, vom presupune că elementele dintr-un container sunt de tipul *TElement* (în loc să folosim întreg sau string sau ceva tip concret).
- În continuare vom vorbi despre diferite containere, dar le vom trata ca *Tipuri Abstracte de Date*.

Tipuri Abstracte de Date - TAD

- Un *Tip Abstract de Date - TAD* este un tip de date (în general definit de utilizator) care are 2 proprietăți:
  - *obiectele* (adică variabilele care sunt de tipul respectiv) pot fi *specificate* (descrise) independent de reprezentarea lor (fără a spune ce câmpuri vor fi folosite pentru implementare)
  - *operațiile* pot fi specificate (prin precondiții și postcondiții) independent de implementarea lor (fără a spune cum vor fi implementate).

Containere ca TAD I

- A prezenta un container ca un TAD, înseamnă a nu vorbi de implementarea containerului.
- De exemplu, eu pot spune că am un container în care elementele sunt unice, fără să spun cum sunt acele elemente stocate exact (fără a spune că le voi pune într-un vector, de exemplu).
- În mod similar, pot spune că un element nou va fi adăugat doar dacă el nu mai există în container, fără a spune cum exact se întâmplă verificarea și adăugarea.

Containere ca TAD II

- De ce nu vorbim despre implementarea unui container?
- Un container reprezintă ceva abstract, nu implică nicio implementare anume.
- Majoritatea containerelor pot fi implementate în mai multe moduri, aceste implementări având 2 caracteristici comune:
  - Toate implementările trebuie să respecte caracteristicile de bază ale containerului (dacă containerul nu poate conține duplicate, atunci nu are voie să conțină duplicate indiferent de modul de implementare)
  - Toate implementările trebuie să respecte interfața containerului (adică să ofere aceeași operații)

Containere ca TAD III

- După ce discutăm de un container ca TAD, vom vorbi și de implementări posibile (prezentarea unui container ca TAD e o chestie abstractă, dacă nu o implementez, nu mă ajută cu nimic, nu îl pot folosi).
- Pentru a implementa un container putem folosi diferite *structuri de date*.

Containere și structuri de date în practică

- Deși par abstracte și teoretice, containerele și structurile de date despre care vom discuta există în majoritatea limbajelor de programare (nu peste tot sunt la fel, și nu toate există în fiecare limbaj de programare, dar ele există).
- Este necesar să cunoaștem specficul pentru fiecare container, ca să le putem folosi corect.

Containere și structuri de date în practică II

- Dacă ele sunt implementate deja, de ce trebuie să vorbim de cum sunt implementate, de ce nu putem pur și simplu să le folosim?

Containere și structuri de date în practică II

- Dacă ele sunt implementate deja, de ce trebuie să vorbim de cum sunt implementate, de ce nu putem pur și simplu să le folosim?
  - Pentru a le folosi cât mai bine, trebuie să înțelegem cum sunt ele implementate.

Containere și structuri de date în practică II

- Dacă ele sunt implementate deja, de ce trebuie să vorbim de cum sunt implementate, de ce nu putem pur și simplu să le folosim?
  - Pentru a le folosi cât mai bine, trebuie să înțelegem cum sunt ele implementate.
  - Unele containere au mai multe implementări, trebuie să înțelegem aceste implementări pentru a alege cea potrivită pentru noi.

Containere și structuri de date în practică II

- Dacă ele sunt implementate deja, de ce trebuie să vorbim de cum sunt implementate, de ce nu putem pur și simplu să le folosim?
  - Pentru a le folosi cât mai bine, trebuie să înțelegem cum sunt ele implementate.
  - Unele containere au mai multe implementări, trebuie să înțelegem aceste implementări pentru a alege cea potrivită pentru noi.
  - Poate la un moment dat lucrăm într-un limbaj de programare unde nu sunt implementate și trebuie să le implementăm noi.

Containere și structuri de date în practică II

- Dacă ele sunt implementate deja, de ce trebuie să vorbim de cum sunt implementate, de ce nu putem pur și simplu să le folosim?
  - Pentru a le folosi cât mai bine, trebuie să înțelegem cum sunt ele implementate.
  - Unele containere au mai multe implementări, trebuie să înțelegem aceste implementări pentru a alege cea potrivită pentru noi.
  - Poate la un moment dat lucrăm într-un limbaj de programare unde nu sunt implementate și trebuie să le implementăm noi.
  - Înțelegerea acestor containere și structuri de date ne ajută dacă avem nevoie să definim noi o structură care seamană cu cele existente.

Colecția

- Gândiți-vă la o aplicație care ar trebui să gestioneze cumpărăturile făcute într-un magazin alimentar. Avem nevoie de un container în care putem reține ceea ce am cumpărat (un fel de coș de cumpărături).
- Care ar fi caracteristicile acestui container?

Colecția

- Gândiți-vă la o aplicație care ar trebui să gestioneze cumpărăturile făcute într-un magazin alimentar. Avem nevoie de un container în care putem reține ceea ce am cumpărat (un fel de coș de cumpărături).
- Care ar fi caracteristicile acestui container?
  - containerul poate să conțină duplicate (pot cumpăra mai multe bucăți din ceva)
  - ordinea elementelor nu contează (e în regulă dacă reții lapte și pâine sau pâine și lapte)

Colecția

- Gândiți-vă la o aplicație care ar trebui să gestioneze cumpărăturile făcute într-un magazin alimentar. Avem nevoie de un container în care putem reține ceea ce am cumpărat (un fel de coș de cumpărături).
- Care ar fi caracteristicile acestui container?
  - containerul poate să conțină duplicate (pot cumpăra mai multe bucăți din ceva)
  - ordinea elementelor nu contează (e în regulă dacă reținlapte și pâine sau pâine și lapte)
- **Colecția** (**Bag** în engleză) este un container în care elementele se pot repeta și elementele nu au o anumită poziție.

Colecția II

- Ce înseamnă că *elementele nu au o anumită poziție*?

Colecția II

- Ce înseamnă că *elementele nu au o anumită poziție*?
  - Operațiile din interfața Colecției nu primesc poziții ca parametru:

Colecția II

- Ce înseamnă că *elementele nu au o anumită poziție*?
  - Operațiile din interfața Colecției nu primesc poziții ca parametru:
    - Nu există adăugare pe o anumită poziție, doar adăugare
    - Nu există ștergerea unui element de pe o poziție, doar ștergerea unui element dat
    - Nu există operație care să returneze un element de pe o poziție

Colecția II

- Ce înseamnă că *elementele nu au o anumită poziție*?
  - Operațiile din interfața Colecției nu primesc poziții ca parametru:
    - Nu există adăugare pe o anumită poziție, doar adăugare
    - Nu există ștergerea unui element de pe o poziție, doar ștergerea unui element dat
    - Nu există operație care să returneze un element de pe o poziție
  - Dacă adăugăm într-o colecție vidă pe rând elementele: ciocolată, lapte, pâine, ciocolată, după care tipărim conținutul colecției, elementele pot fi tipărite în orice ordine:
    - ciocolată, lapte, pâine, ciocolată
    - ciocolată, ciocolată, lapte, pâine
    - lapte, ciocolată, pâine, ciocolată
    - etc.

Colecția III

- Evident, în momentul în care implementez o colecție, elementele vor fi stocate undeva în memorie, dar ordinea în care sunt stocate nu este importantă, și utilizatorul care folosește o colecție (gata implementată) nu trebuie să aibă nicio presupunere legată de ordinea elementelor.

Colecția - Interfață

- *Interfața* unei colecții este lista operațiilor - împreună cu precondiții și postcondiții - care trebuie să existe pentru o colecție.
  - precondiții - condiții care trebuie să fie adevărate înainte de a apela o funcție
  - postcondiții - condiții care trebuie să fie adevărate după ce funcția se termină
- Interfața este de multe ori considerată ca un contract: dacă un TAD vrea să fie colecție, trebuie să conțină operațiile enumerate în interfață.
- Ce operații ar trebui să conțină interfața unei Colecții?

Colecția - Interfață I

- creează ()
  - **descriere:** creează o colecție vidă (constructorul clasei Colectie)
  - **pre:** adevărat
  - **post:** *colecția* a fost creată, este o Colecție vidă

Colecția - Interfață II

- col.adaugă (e)
  - **descriere:** adaugă un element în colecție
  - **pre:** *col* este o colecție, *e* este un TElement
  - **post:** *e* a fost adăugat în colecție

Colecția - Interfață III

- col.șterge(e)
  - **descriere:** șterge o apariție a elementului din colecție (dacă elementul apare de mai multe ori, o apariție este ștearsă, dacă nu apare deloc, colecția rămâne nemodificată)
  - **pre:** *col* este o colecție, *e* este un TElement
  - **post:** o apariție a lui *e* a fost ștearsă din *col*

Colecția - Interfață IV

- col.dim()
  - **descriere:** returnează numărul de elemente din colecție
  - **pre:** *col* este o colecție
  - **post:** returnează nr de elemente din *col*

Colecția - Interfață V

- col.caută(e)
  - **descriere:** verifică dacă un element apare într-o colecție
  - **pre:** *col* este o colecție, *e* este un TElement
  - **post:** returnează adevărat dacă *e* apare în *col*, fals altfel

Colecția - Interfață VI

- ??? (vom discuta mai târziu)
  - **descriere:** ???
  - **pre:** ???
  - **post:** ???

Colecția - Interfață VII

- Lista anterioară reprezintă lista minimă de operații care trebuie să existe pentru o colecție.
- Alte operații posibile:
  - șterge toate elementele din colecție (golește colecția)
  - returnează numărul de apariții pentru un element
  - verifică dacă colecția este vidă

Colecția în Java

- Java are un pachet, numit *collections* unde există implementări de containere.
- În acest pachet nu există implementare pentru *colecție*.

Colecția - exemplu de utilizare

- Să considerăm problema următoare: *să reprezentăm conținutul unui portmoneu. La început se citesc pe rând valorile bancnotelor din portmoneu (până se citește valoarea 0). Afișați suma totală pe care o avem.*

Colecția - exemplu de utilizare

- Să considerăm problema următoare: *să reprezentăm conținutul unui portmoneu. La început se citesc pe rând valorile bancnotelor din portmoneu (până se citește valoarea 0). Afișați suma totală pe care o avem.*
- Cum bancnotele se pot repeta, iar ordinea lor nu contează, colecția este un container potrivit pentru a rezolva această problemă.

Colecția - exemplu de utilizare

- Să considerăm problema următoare: *să reprezentăm conținutul unui portmoneu. La început se citesc pe rând valorile bancnotelor din portmoneu (până se citește valoarea 0). Afișați suma totală pe care o avem.*
- Cum bancnotele se pot repeta, iar ordinea lor nu contează, colecția este un container potrivit pentru a rezolva această problemă.
- În momentul în care folosim containerul colecție pentru a rezolva o problemă, folosim doar *interfața* colecției, doar operațiile, nu ne gândim la reprezentare (momentan nici nu știm nimic despre reprezentare).

Colecția - exemplu de utilizare II

- Vom împărți rezolvarea problemei în 3 subprograme:

Colecția - exemplu de utilizare II

- Vom împărți rezolvarea problemei în 3 subprograme:
  - Un subprogram care citește bancnotele (bancnotele sunt reprezentate de numere, valoarea bancnotelor, de ex. 5, 10, 50, etc.)
  - Un subprogram care calculează suma totală
  - Programul principal

Colecția - exemplu de utilizare - citire portmoneu

```
subalgoritm citesteBancnote(portmoneu: Colectie) este:
    val: întreg
    val = 1
    câttimp val ≠ 0 execută
        scrie "Valoarea bancnotei:"
        citește val
        dacă val ≠ 0 atunci //să nu adăugăm ultimul 0
            portmoneu.adaugă(val)
        sf_dacă
    sf_câttimp
sf_subalgoritm
```

Colecția - exemplu de utilizare - citire portmoneu

```
subalgoritm citesteBancnote(portmoneu: Colectie) este:
    val: întreg
    val = 1
    câttimp val ≠ 0 execută
        scrie "Valoarea bancnotei:"
        citește val
        dacă val ≠ 0 atunci //să nu adăugăm ultimul 0
            portmoneu.adaugă(val)
        sf_dacă
    sf_câttimp
sf_subalgoritm
```

- Am discutat că tipurile definite de utilizator (cum e Colecția) se transmit prin referință, deci orice modificare făcută pe variabila *portmoneu* (de exemplu, adăugări de elemente) este vizibilă și în afara funcției.

Colecția - exemplu de utilizare - calcul suma totală

- Cum calculăm suma totală de bani din portmoneu?

Colecția - exemplu de utilizare - calcul suma totală

- Cum calculăm suma totală de bani din portmoneu?
- Dacă ne uităm la operațiile discutate până acum pentru colecție, putem observa că nu avem nicio operație cu care să *vedem* ce elemente avem în colecție.

Colecția - exemplu de utilizare - calcul suma totală II

- Cel care folosește colecția are nevoie de o metodă de a *parcurge* elementele colecției (nu contează în ce ordine, dar trebuie să existe o metodă de a parcurge toate elementele).

Colecția - exemplu de utilizare - calcul suma totală II

- Cel care folosește colecția are nevoie de o metodă de a *parcurge* elementele colecției (nu contează în ce ordine, dar trebuie să existe o metodă de a parcurge toate elementele).
- Cel care a implementat colecția vrea să ascundă reprezentarea colecției (ce câmpuri a folosit pentru reprezentare), pune la dispoziția utilizatorului doar operațiile colecției.

Colecția - exemplu de utilizare - calcul suma totală II

- Cel care folosește colecția are nevoie de o metodă de a *parcurge* elementele colecției (nu contează în ce ordine, dar trebuie să existe o metodă de a parcurge toate elementele).
- Cel care a implementat colecția vrea să ascundă reprezentarea colecției (ce câmpuri a folosit pentru reprezentare), pune la dispoziția utilizatorului doar operațiile colecției.
- Soluția pentru a rezolva acest conflict este folosirea unui **iterator**.
- Deși noi am început să discutăm despre iterator în legătură cu colecția, el există pentru mai multe containere.

Iterator - în general

- *Iteratorul* este un *tip abstract de date - TAD*, implementat separat de container, folosit pentru a parcurge (itera) elementele unui container.

Iterator - în general

- *Iteratorul* este un *tip abstract de date - TAD*, implementat separat de container, folosit pentru a parcurge (itera) elementele unui container.
- Pentru a ascunde de la utilizator reprezentarea exactă a unui container (adică acele câmpuri folosite), dar în același timp a da posibilitatea de a itera elementele containerului, se folosește iteratorul. Iteratorul este o structură care *cunoaște* reprezentarea containerului și conține un *element curent* din container.
- Din moment ce iteratorul cunoște reprezentarea containerului, e treaba noastră să implementăm și iteratorul când implementăm un container (de ex. când implementăm Colecția, trebuie să implementăm și iterator pentru Colecție).

Iterator - în general

- Iteratorul are un element curent (din containerul iterat), și are o operație pentru a returna acest element. Iteratorul mai are o operație cu care se poate trece la următorul element curent.
- A parcurge elementele unui container folosind iteratorul se face folosind aceste operații (și încă ceva, ce vom discuta mai târziu):
  - Returnează elementul curent...
  - Du-te la următorul element
  - Returnează elementul curent ...
  - Du-te la următorul element
  - ...

Iterator - Interfața I

- Iteratorul fiind un TAD separat, are propria interfață - adică lista de operații care trebuie să fie implementate de un iterator.
- Am discutat că iteratorul este creat pentru un container (nu există iterator fără un container) și că reține un element curent din container.

Iterator - Interfața II

- creeaza (cont)
  - **descriere:** creează un iterator nou pentru un container (constructorul clasei Iterator). Este obligatoriu să furnizăm containerul pentru care se creează iteratorul!
  - **pre:** *cont* reprezintă un container
  - **post:** creează un iterator pentru containerul *cont*. Elementul curent din *it* este "primul" element din *cont*

Iterator - Interfața III

- it.element()
  - **descriere:** funcția element returnează elementul curent din iterator
  - **pre:** *it* este un iterator valid
  - **post:** returnează elementul curent din *it*

Iterator - Interfața IV

- it.următor()
  - **descriere:** trece la următorul element din container (adică elementul curent din iterator devine următorul element din container. Dacă nu mai sunt elemente neparcurse în container, iteratorul devine invalid)
  - **pre:** *it* este un iterator valid
  - **post:** elementul curent din *it* devine următorul element din container sau *it* devine invalid.

Iterator - Interfața V

- it.valid()
  - **descriere:** verifică dacă elementul curent din iterator este un element valid
  - **pre:** *it* este un iterator
  - **post:** returnează adevărat, dacă elementul curent din *it* este valid, fals altfel

Iterator - Interfața VI

- Aceste 4 operații sunt cele de bază care trebuie să existe pentru un iterator. Un iterator care are doar aceste operații se numește iterator *unidirecțional* - se deplasează doar într-o singură direcție, înainte -, și *read-only* - nu modifică containerul, doar returnează elemente din el.

Iterator - Interfața VII

- Există și iterator *bi-direcțional* - iterator care se poate deplasa în ambele direcții.
  - Pe lângă operația *următor*, un iterator bi-direcțional are și operație *anterior* (unde elementul curent din iterator devine elementul de dinaintea elementului curent).
  - Putem să avem operație *prim* - care setează iteratorul pe primul element din container - și *ultim* - care setează iteratorul pe ultimul element din container -.

Iterator - Interfața VIII

- Există și iterator *read-write* - iterator care poate să modifice containerul.
  - Poate să aibă operație care să înlocuiască elementul curent cu un alt element.
  - Poate să aibă operație care să insereze un element nou înainte de elementul curent (sau după elementul curent, în funcție de implementare).
  - Poate să aibă operație care șterge elementul curent.

Iterator - Interfața IX

- Noi în continuare vom discuta/implementa/folosi doar iterator uni-direcțional și read-only.

Utilizarea iteratorului

- Orice container care poate fi iterat (există și câteva containere care nu pot avea iterator) trebuie să aibă în interfață o operație numită *iterator* care returnează un iterator pentru containerul respectiv.

Utilizarea iteratorului

- Orice container care poate fi iterat (există și câteva containere care nu pot avea iterator) trebuie să aibă în interfață o operație numită *iterator* care returnează un iterator pentru containerul respectiv.

- După ce avem iteratorul, parcurgerea containerului se face folosind operațiile iteratorului:
  - Apelăm funcția *element* să avem elementul curent.
  - Facem ce avem de făcut cu elementul curent (îl afișăm, îl adunăm la sumă, etc.).
  - Apelăm funcția *următor* ca să trecem la următorul element.
  - Apelăm funcția *valid*, pentru a vedea dacă mai avem element valid. Dacă da, apelăm iar *element* și *următor*. Dacă valid îmi returnează fals, am terminat parcurgerea.

Utilizarea iteratorului

- Deci, parcurgerea unui container care are iterator poate fi făcută în modul următor:

```
//cont - reprezintă orice container care poate fi iterat
it: Iterator
it = cont.iterator() //apelăm funcția iterator pentru a crea un interator
câttimp it.valid() execută //începem parcurgerea
    elemCurent: TElement
    elemCurent = it.element() //accesăm elementul curent
    @ facem ceva cu elemCurent
    it.următor() //trecem la elementul următor
sf_câttimp
```

Utilizarea iteratorului

- Deci, parcurgerea unui container care are iterator poate fi făcută în modul următor:

```
//cont - reprezintă orice container care poate fi iterat
it: Iterator
it = cont.iterator() //apelăm funcția iterator pentru a crea un interator
câttimp it.valid() execută //începem parcurgerea
    elemCurent: TElement
    elemCurent = it.element() //accesăm elementul curent
    @ facem ceva cu elemCurent
    it.următor() //trecem la elementul următor
sf_câttimp
```

- **Parcurgerea cu un iterator se face în acest mod indiferent de containerul exact și indiferent de reprezentarea containerului!**

Iteratorul în Java

- Iteratorul există și în limbajul de programare Java, numai arată puțin diferit.
- Documentația pentru Iterator se găsește la adresa https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

Colecția - exemplu de utilizare - calcul suma totală

- Pentru a calcula suma bancnotelor din portmoneu, vom folosi un iterator ca să parcurgem colecția

```
funcție sumaTotală(portmoneu: Colecție) este:
    //portmoneu este o colecție în care avem valorile bancnotelor
    suma, bancnota: întreg
    suma = 0 //aici vom calcula suma
    iter: Iterator
    iter =portmoneu.iterator() //creem un iterator pe portmoneu
    câttimp iter.valid() execută
        bancnota = iter.element()
        suma = suma + bancnota
        iter.urmator()
    sf_câttimp
    returnează suma //returnăm suma
sf_funcție
```

Colecția - exemplu de utilizare - program principal

- Ne mai trebuie programul principal care să apeleze funcțiile făcute și să afișeze suma totală

algoritm BaniiMei **este:**
  portmoneu: Colecție *//definesc o variabilă de tip Colecție*
  citesteBancnote(portmoneu)
  suma: întreg
  suma = sumaTotală(portmoneu)
  scrie "Suma totală în portmoneu este: " + suma
sf_algoritm

TAD Mulțime

- Să considerăm problema următoare: *Pentru a evita ca cineva să voteze de mai multe ori la alegeri, se implementează un sistem care reține într-un container CNP-ul persoanelor care au votat deja.*
- Care ar trebui să fie caracteristicile acestui container?

TAD Mulțime

- Să considerăm problema următoare: *Pentru a evita ca cineva să voteze de mai multe ori la alegeri, se implementează un sistem care reține într-un container CNP-ul persoanelor care au votat deja.*
- Care ar trebui să fie caracteristicile acestui container?
  - Nu contează ordinea elementelor (nu contează cine în ce ordine a votat)
  - Elementele trebuie să fie unice (CNP-urile nu au voie să se repete).

TAD Mulțime

- Containerul care se potrivește când avem nevoie să reținem elemente care sunt unice (nu se repetă) și care nu au o poziție, se numește **Mulțime**.

- TAD-ul **Mulțime** (numit **Set** în engleză) seamănă mult cu TAD-ul Colecție cu excepția că într-o mulțime elementele sunt unice, dar într-o colecție elementele se pot repeta.

Mulțime - interfață

- Ce operații ar trebui să aibă TAD Mulțime?

Mulțime - Interfață I

- creează ()
  - **descriere:** creează o mulțime vidă
  - **pre:** adevărat
  - **post:** *mulțime* a fost creat, este o Mulțime vidă

Mulțime - Interfață II

- mul.adaugă(e)
  - **descriere:** adaugă un element în mulțime (dacă nu există)
  - **pre:** *mul* este o Mulțime, *e* este un TElement
  - **post:** *e* a fost adăugat în mulțimea *mul* numai dacă înainte nu a existat în ea (nu se mai adaugă elementul încă o dată)

Mulțime - Interfață III

- mul.șterge(e)
  - **descriere:** șterge un element dat din mulțime
  - **pre:** *mul* este o Mulțime, *e* este un TElement
  - **post:** *e* a fost șters din *mul* (dacă *e* nu era în *mul*, *mul* rămâne neschimbată)

Mulțime - Interfață IV

- mul.dim()
  - **descriere:** returnează numărul de elemente din mulțime
  - **pre:** *mul* este o mulțime
  - **post:** dim returnează nr de elemente din *mul*

Mulțime - Interfață V

- mul.caută(e)
  - **descriere:** verifică dacă un element apare într-o mulțime
  - **pre:** *mul* este o Mulțime, *e* este un TElement
  - **post:** caută returnează adevărat dacă *e* apare în *mul*, fals altfel

Mulțime - Interfață VI

- mul.iterator()
  - **descriere:** returnează un iterator pentru o mulțime
  - **pre:** *mul* este o Mulțime
  - **post:** iterator returnează un iterator pe mulțimea *mul*

Mulține în Java

- În Java există containerul Mulțime, se numește **Set**
- Lista operațiilor pentru **Set** se găsește la adresa: https://docs.oracle.com/javase/7/docs/api/java/util/Set.html
- Sunt mai multe operații decât cele discutate de noi, dar ce am discutat noi, apare în listă .

Mulțime - exemplu de utilizare

- Să considerăm din nou problema pe care am rezolvat-o folosind colecția: *să reprezentăm conținutul unui portmoneu. La început se citesc pe rând valorile bancnotelor din portmoneu (până se citește valoarea 0). Afișați suma totală pe care o avem în portmoneu.*

- Poate fi rezolvată problema folosind o mulțime?

Mulțime - exemplu de utilizare

- Să considerăm din nou problema pe care am rezolvat-o folosind colecția: *să reprezentăm conținutul unui portmoneu. La început se citesc pe rând valorile bancnotelor din portmoneu (până se citește valoarea 0). Afișați suma totală pe care o avem în portmoneu.*

- Poate fi rezolvată problema folosind o mulțime?

- În varianta curentă nu putem rezolva problema folosind o mulțime, pentru că elementele noastre nu sunt unice. Dar dacă modificăm un pic cerința, ca să reținem și seria bancnotelor, nu doar valoarea lor, putem folosi o mulțime.

Mulțime - exemplu de utilizare

- Vom defini un tip nou, care să reprezinte o bancnotă. Fiecare bancnotă este caracterizată de seria (un string) și de valoarea bancnotei (un întreg).

Bancnotă:
serie: String
valoare: întreg

Mulțime - exemplu de utilizare

- Din nou vom împărți rezolvarea problemei în 3 subprograme:
  - Un subprogram care citește bancnotele
  - Un subprogram care calculează suma totală
  - Programul principal

Mulțime - exemplu de utilizare - citire portmoneu

```
subalgoritm citesteBancnote(portmoneu: Mulțime) este:
    val: întreg
    val = 1
    câttimp val ≠ 0 execută
        scrie "Valoarea bancnotei:"
        citește val
        dacă val ≠ 0 atunci //să nu adăugăm ultimul 0
            serie: String
            scrie "Seria bancnotei:"
            citește serie
            b: Bancnotă
            b.serie = serie
            b.valoare = val
            portmoneu.adaugă(b)
        sf_dacă
    sf_câttimp
sf_subalgoritm
```

Mulțime - exemplu de utilizare

- Cum calculăm suma bancnotelor din portmomenu?

Mulțime - exemplu de utilizare

- Cum calculăm suma bancnotelor din portmomenu?
- Nu avem poziții nici aici, dar, ca la colecție, avem iterator. Vom folosi iteratorul pentru a parcurge mulțimea și a aduna valoarea bancnotelor.

Mulțime - exemplu de utilizare - sumă

funcție sumaTotală(portmoneu: Mulțime) **este**:
  //portmoneu este o mulțime în care avem bancnote
  suma: întreg
  b: Bancnotă
  suma = 0 *//aici vom calcula suma*
  iter: Iterator
  iter =portmoneu.iterator() *//creem un iterator pe portmoneu*
  câttimp iter.valid() **execută**
    b = iter.element() *//b este o bancnotă întreagă*
    suma = suma + b.valoare *//adunăm doar valoarea*
    iter.urmator()
  sf_câttimp
  returnează suma *//returnăm suma*
sf_funcție

Mulțime - exemplu de utilizare - program principal

- Ne mai trebuie programul principal care să apeleze funcțiile făcute și să afișeze suma totală

algoritm BaniiMei **este:**
  portmoneu: Mulțime *//definesc o variabilă de tip Mulțime*
  citesteBancnote(portmoneu)
  suma: întreg
  suma = sumaTotală(portmoneu)
  scrie "Suma totală în portmoneu este: " + suma
sf_algoritm