

9. COLECTII. TIPURI DE DATE SI METODE PARAMETRIZATE (GENERIC)

9.1. Colectii.....	<u>2</u>
9.1.1. Principii.....	<u>2</u>
9.1.2. Modalitati de stocare a obiectelor in interiorul colectiei.....	<u>2</u>
9.1.3. Ierarhia de interfete colectie.....	<u>2</u>
9.1.4. De la interfata la implementare: ...pana la urma ce instantiem?.....	<u>3</u>
9.1.5. API-ul interfețelor colectie.....	<u>3</u>
9.1.5.1. Interfata Collection.....	<u>3</u>
9.1.5.2. Colectii de tip lista: interfata List.....	<u>4</u>
9.1.5.3. Colectii de tip multime.....	<u>4</u>
9.1.5.3.1. Interfata Set.....	<u>4</u>
9.1.5.3.2. Interfata SortedSet.....	<u>4</u>
9.1.5.3.3. Interfata NavigableSet.....	<u>4</u>
9.1.5.4. Colectii de tip coada.....	<u>5</u>
9.1.5.4.1. Interfata Queue.....	<u>5</u>
9.1.5.4.2. Interfata Deque.....	<u>5</u>
9.1.5.5. Colectii de tip set de corespondente.....	<u>5</u>
9.1.5.5.1. Interfata Map.....	<u>5</u>
9.1.5.5.2. Interfata SortedMap.....	<u>5</u>
9.1.5.5.3. Interfata NavigableMap.....	<u>6</u>
9.1.6. Parcursarea unei colectii.....	<u>6</u>
9.1.6.1. Modalitati.....	<u>6</u>
9.1.6.2. Bucla for (pentru colectii de tip List).....	<u>6</u>
9.1.6.3. Bucla for-each.....	<u>6</u>
9.1.6.4. Iteratori.....	<u>7</u>
9.1.7. Operatii cu colectii: clasa Collections.....	<u>7</u>
9.1.8. Ordonarea obiectelor.....	<u>8</u>
9.1.8.1. Principii.....	<u>8</u>
9.1.8.2. Ordonarea dupa criteriul default (“native sorting”.....	<u>9</u>
9.1.8.3. Ordonarea dupa criterii aditionale.....	<u>9</u>
9.1.9. Interfata Enumeration.....	<u>10</u>
9.2. Generics: tipuri de date si metode parametrizate.....	<u>10</u>
9.2.1. Concepte.....	<u>10</u>
9.2.2. Mecanismul de functionare al parametrizarii.....	<u>11</u>
9.2.3. Sintaxa de definire si utilizare a tipurilor de date si metodelor parametrizate.....	<u>12</u>
9.2.3.1. Ce forme iau si unde pot fi utilizati parametrii-tip?.....	<u>12</u>
9.2.3.2. Tipuri de date parametrizate.....	<u>12</u>
9.2.3.2.1. Sintaxa de definire.....	<u>12</u>
9.2.3.2.2. Sintaxa de utilizare - invocarea de tipuri parametrizate.....	<u>13</u>
9.2.3.3. Metode si constructori parametrizati.....	<u>13</u>
9.2.3.3.1. Sintaxa de definire.....	<u>13</u>
9.2.3.3.2. Sintaxa de utilizare.....	<u>14</u>
9.2.3.4. Deducerea automata a valorii unui parametru-tip (type inference).....	<u>15</u>
9.2.4. Problema relatiilor intre tipuri de date parametrizate.....	<u>16</u>
9.2.5. Parametri-tip ce folosesc wildcard-uri.....	<u>17</u>
9.2.6. Interoperabilitatea cu cod care nu foloseste tipuri parametrizate.....	<u>17</u>
9.3. BIBLIOGRAFIE.....	<u>18</u>

9.1. Colectii

9.1.1. Principii

Deseori avem nevoie sa lucram cu un grup de obiecte folosit ca o entitate de sine statatoare. Am intalnit acest lucru atunci cand am definit tablouri (fie ele de primitive sau de obiecte) – care reprezinta in fond un set de variabile fara nume propriu dar accesibile printr-un nume si un index. In cele ce urmeaza vom descoperi ca tabloul este doar una din modalitatatile de a realiza un container pentru obiecte.

O colectie, la modul general, este un obiect ce contine un grup de obiecte de orice tip, si care ne pune la dispozitie operatiuni de adaugare/eliminare/cautare/parcursere etc. a elementelor (setul de operatiuni posibile variaza in functie de tipul colectiei). JDK ofera Java Collections Framework, ce contine o infrastructura de interfete si clase concrete de ase natura gandita incat sa ofere programatorului flexibilitate maxima, el putand alege atat modul in care sunt stocate datele, cat si modul in care le manipuleaza si facilitatile dorite (pastrarea ordinii obiectelor, ordonarea acestora, interzicerea duplicatelor etc.).

9.1.2. Modalitati de stocare a obiectelor in interiorul colectiei

Modul in care sunt stocate efectiv datele in interiorul colectiei determina performantele acesteia. Iata cateva modalitati de reprezentare interna a grupului de obiecte in cadrul colectiei, intalnite in clasele din Java Collections Framework:

- **tablou de obiecte** – obiectele din cadrul colectiei sunt memorate sub forma unui membru privat de tip array de obiecte. Implementarea cu tablou ofera urmatoarele caracteristici:
 - avantaj: viteza mare de accesare a elementelor. Deoarece tabloul ocupa o zona contigua de memorie si este format din elemente de acelasi tip, timpul necesar pentru accesarea unui element este constant: trebuie doar inmultita pozitia elementului cu dimensiunea unui element si rezultatul adunat la adresa de incepere a tabloului in memorie
 - dezavantaj: este costisitor pentru adaugarea/stergerea/inserarea de obiecte din doua motive: 1) lungimea fixa, immutable (care implica crearea unui nou tablou si copierea informatiei din cel vechi cand tabloul se umple) si 2) mutarile de elemente ce trebuie efectuate la stergere si introducere. Spre exemplu, stergand elementul de pe pozitia 3, toate elementele pana la finalul tabloului trebuie mutate cu o pozitie la stanga
- **lista simplu sau dublu inlantuita** – colectia este formata dintr-o serie de obiecte element, dintre care unul este desemnat ca fiind cap de lista. Fiecare element contine o referinta catre cel urmator si eventual cel precedent (pentru liste dublu inlantuite). Parcurserea incepe de la capatul de lista si presupune avansarea din element in element. Caracteristicile acestei solutii sunt urmatoarele:
 - dezavantaj: accesul la un element depinde liniar de pozitia acestuia, deoarece trebuie parcursa lista din element in element pana la pozitia dorita
 - avantaj: operatiile de introducere/stergere de element presupun numai reorientarea unui numar finit de referinte. Spre exemplu, la stergerea unui element trebuie doar reconectat elementul anterior lui cu cel ulterior
- **arbore binar** – obiectele sunt memorate ordonat sub forma unui arbore binar, fiind plasate in arbore dupa niste reguli de ase natura incat arborele sa ramana permanent echilibrat. Caracteristici:
 - avantaje:
 - obiectele sunt deja ordonate
 - timp de acces la un element mai mic decat in cadrul unei liste. Daca pentru o lista, in cel mai rau caz, trebuie sa o parcurgem pe toata ca sa ajungem la elementul dorit, arborele are o adancime mult mai mica si deci accesul la un element presupune mai putine operatii. (Nota: observam ca cel mai inefficient arbore este cel cu o singura ramura - care este de fapt o lista!)
 - dezavantaj: esfertul ordonarii se distribuie in timp, si mai exact introduce o penalitate de viteza la adaugarea/stergerea fiecarui obiect. Obiectul de introdus trece prin niste comparatii succesive pana "isi gaseste locul" in arbore, ceea ce face introducerea costisitoare; prin analogie, la stergerea unui obiect este necesara ocazionala reechilibrarea arborului, si deci din nou penalitate de viteza
- **tabel de dispersie (hash table)** – fiecare obiect al colectiei este pus in corespondenta hash code-ul sau, iar obiectele sunt grupate in sub-categorii in functie de hash code, formandu-se atat de grupuri de obiecte cate hash code-uri distincte exista in colectie. Cand se cauta un obiect in cadrul colectiei este identificat intai grupul sau pe

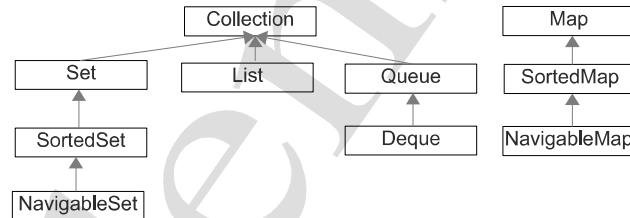
baza hash code-ului, iar apoi in cadrul grupului se efectueaza cautare secventiala. In acest fel se evita cautarea secventiala in intreaga colectie, care este costisitoare. Caracteristicile implementarii:

- avantaj: cautare foarte rapida in cantitati mari de obiecte, in masura in care metoda hashCode() a obiectelor este implementata de asa natura incat sa distribuie obiectele cat mai uniform in grupuri
- dezavantaje:
 - cantitatea mai mare de memorie ocupata pentru a tine evidenta grupurilor de obiecte
 - ordine impredictibila a obiectelor la parcurgerea colectiei (iterand de doua ori prin colectie, vom obtine succesiuni diferite ale elementelor)

9.1.3. Ierarhia de interfete colectie

Java pune la dispozitia programatorului *Collections Framework* – un set de clase destinate reprezentarii si manipularii colectiilor de obiecte, oricare ar fi tipurile de date continue.

Intreaga arhitectura porneste de la interfetele din figura alaturata, grupate in doua ierarhii. Aceste interfete specifica moduri generale de manipulare a colectiei, implementarea efectiva cazand in sarcina claselor concrete.



Iata o scurta descriere a scopului fiecarei interfete:

- **Collection** - este cea mai generala colectie, care nu impune niciun fel de restrictii asupra elementelor sale
 - **Set** - modeleaza conceptul matematic de multime. Nu mentine ordinea elementelor si nu permite duplicate
 - **SortedSet** - un Set care mentine elementele ordonate conform unei reguli specificate de programator
 - **NavigableSet** - un SortedSet cu metode suplimentare pentru parcurgerea in ambele sensuri a elementelor colectiei
 - **List** - implementeaza o succesiune de elemente, disponand de conceptul de pozitie a unui element. permite ordonarea elementelor. Poate contine elemente dupicate
 - **Queue** - modeleaza o coada de elemente, oferind cateva metode specifice suplimentare fata de collection
 - **Deque** - modeleaza o coada cu doua capete, in care elementele pot fi introduse/eliminate atat la inceputul cat si la sfarsitul cozii
- **Map** - o colectie de perechi cheie-valoare, in care atat cheia, cat si valoarea sunt obiecte. Fiecare valoare poate fi accesata numai pe baza cheii corespondente
 - **SortedMap** - un Map care mentine elementele ordonate dupa chei, dupa o regula specificata de catre programator
 - **NavigableMap** - un SortedMap cu metode suplimentare pentru parcurgerea si localizarea obiectelor

9.1.4. De la interfata la implementare: ...pana la urma ce instantiem?

In viata reala programatorul trebuie sa poata alege, in mod independent, urmatoarele doua aspecte ale colectiei dorite: ce facilitati are colectia dorita si care este implementarea ei interna. In Java Collections Framework aceasta inseamna intersecția dintre o anumita interfata dintre cele prezentate anterior si o anumita implementare interna. Incrucisările de acest fel se regasesc sub forma de clase concrete predefinite, prezentate in tabelul de mai jos.

Numele fiecarei clase este de forma <Implementare><Interfata>. De exemplu, *ArrayList* este o lista care stocheaza intern obiectele sub forma unui tablou. Iata o scurta descriere a fiecarei clase:

- **HashSet** – un Set construit pe baza de hashtable. Este rapid, dar nu garantza ordinea obiectelor la iterare
- **LinkedHashSet** – la fel ca precedentul, insa elementele sunt in plus dublu-inlantuite printre-o lista. Efectul este pastrarea ordinii de insertie a elementelor si deci o ordine de iterare predictibila (ceea ce nu se intampla in cazul lui HashSet).

Interfata	Implementari cu hash table	Implementari cu tablou	Implementari cu arbore	Implementari cu lista	Implementari combinate (hash table + lista)
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

Observatie: Aceasta clasa NU implementeaza *List*, ci doar foloseste intern o structura de tip lista pentru pastrarea ordinii elementelor.

- **TreeSet** – Set a carui implementare se bazeaza pe un arbore binar. Mai lent, insa elementele sunt pastrate automat ordonate ascendent. Clasa dispune de un constructor ce primește ca parametru un *Comparator* pentru specificarea regulilor de ordonare a obiectelor
- **ArrayList** – lista de obiecte in care stocarea acestora se face intr-un tablou redimensionabil; are viteza mare de acces, insa este nerecomandata pentru liste in care se efectueaza des operatiuni de stergere/inserare la inceputul listei
- **LinkedList** – lista dublu inlantuita; obiectele sunt pastrate ordonat, si pot fi parcurse in oricare din cele doua sensuri. Putem accesa indexat oricare dintre obiectele listei, putem adauga, sterge si insera obiecte pe orice pozitie (vezi metodele clasei *List*)
- **HashMap** – un tabel cu echivalente cheie-valoare; atat cheia cat si valoarea sunt obiecte. Nu sunt permise duplicate ale cheii. Un container rapid, dar care nu mentine ordinea elementelor (ordinea de iterare este imprevedibilă).
- **TreeMap** – la fel ca mai sus dar bazat pe un arbore binar. Viteza mai mica dar elementele sunt ordonate ascendent in ordinea naturala a cheii

9.1.5. API-ul interfetelor colectie

9.1.5.1. Interfata Collection

Reprezinta cel mai general container pentru obiecte; nu impune nici un fel de reguli – obiectele pot fi de orice tip, nu sunt mentinute in ordine si pot exista duplicate.

Metode cu care se manipuleaza colectia:

- operatii elementare: **size()**, **isEmpty()**, **contains(Object)**, **add(Object)**, **remove(Object)**
- operatii inter-colectii: **containsAll(Collection)**, **addAll(Collection)**, **removeAll(Collection)**, **retainAll(Collection)**
- reprezentare ca tablou: **toArray()**

Nota: continutul unei colectii poate fi afisat direct prin intermediul lui *System.out.println()*.

Functionalitatea de adaugare si stergere pentru clasele concrete care implementeaza Collection este optionala (in caz ca operatia nu este disponibila, metodele corespunzatoare genereaza *UnsupportedOperationException*). In acest fel se lasa libertatea claselor concrete de a crea implementari de colectie immutable.

9.1.5.2. Colectii de tip lista: interfata List

Modeleaza o secventa de obiecte; ordinea in care sunt introduce obiectele conteaza si se mentine (de unde rezulta ca, la acest tip de colectie, are sens ideea de pozitie a unui obiect si de acces aleator la obiecte folosind pozitia). Sunt permise duplicatele.

Metodele suplimentare fata de Collection reprezinta operatiile pe care le face posibile existenta index-ului fiecarui obiect:

- adaugare/eliminare de obiect: **get(int pozitie)**, **set(int pozitie, Object)**, **add(int pozitie, Object)**, **remove(int pozitie)**
- determinarea pozitiei unui obiect in cadrul colectiei: **indexOf(Object)**, **lastIndexOf(Object)**

9.1.5.3. Colectii de tip multime

9.1.5.3.1. Interfata Set

Reprezinta o colectie in care *nu sunt permise duplicatele*; modeleaza conceptul matematic de multime.

Setul de metode este cel mostenit de la Collection. Clasele concrete ce implementeaza Set trebuie sa respecte “contractul” interfetei (desi nu pot fi obligate de catre compilator sau masina virtuala).

Metoda `add(Object)` returneaza true daca colectia a fost modificata in urma apelarii metodei si false in caz contrar. In acest fel putem afla daca un obiect proaspat adaugat a fost sau nu duplicat.

9.1.5.3.2. Interfata SortedSet

Aceasta interfata modeleaza o multime in care elementele sunt memorate in ordinea lor naturala sau intr-o una specificata de catre programator prin intermediul unui obiect de tip *Comparator* (vezi capitolul despre ordonarea obiectelor).

Operatiuni suplimentare fata de Set:

- extragerea obiectelor “mai mici”/“mai mari” decat cel specificat: `headSet(Object o)` si `tailSet (Object o)`
- extragerea primului si ultimului element: metodele `first()` si `last()`
- extragerea unui interval de obiecte
 - constuit din toate elementele aflate dupa un anumit element din multime: `tailSet(Object dupa)`
 - specificat prin obiectele de inceput si de final ale intervalului: `Set subSet(Object inceput, Object final)`

9.1.5.3.3. Interfata NavigableSet

Aceasta interfata modeleaza un SortedSet cu capabilitati suplimentare, precum:

- aflarea elementului imediat urmator sau anterior unui anumit obiect: `ceiling(Object)` → elementul imediat urmator care este mai mare sau egal cu cel specificat; `floor(Object)` → analog pentru elementul anterior
- aflarea celui mai apropiat elementul care este strict mai mic sau mai mare decat cel specificat: `lower(element)` si `higher(element)`
- parcurgerea in ambele sensuri a elementelor: metoda `descendingIterator()` (vezi sectiunea despre iteratori)

9.1.5.4. Colectii de tip coada

9.1.5.4.1. Interfata Queue

O colectie de tip *Queue* implementeaza o coada de elemente, utilizata in general pentru a stoca obiecte in vederea procesarii. Elementele patrund printr-un capat al cozii si sunt eliminate din celalalt. Coada poate pastra ordinea de adaugare a elementelor sau le poate reordona, prioritizand anumite elemente in functie de reguli specificate de catre programator. Fata de o colectie obisnuita, o coada ofera metode pentru:

- adaugarea unui nou element la capatul cozii: metodele `add(element)` si `offer(element)`. Diferenta dintre ele este ca prima arunca o exceptie in caz de eroare, pe cand a doua returneaza false
- eliminarea primului element din coada (cel curent): `remove()` si `poll()`. Diferenta intre metode este ca, in caz de eroare, `remove()` arunca exceptie iar `poll()` produce `null`
- accesarea elementului curent (primul din coada): `element()` si `peek()`. In cazul in care coada este goala, prima metoda arunca exceptie iar a doua produce `null`

9.1.5.4.2. Interfata Deque

O colectie de tip *Deque* implementeaza o coada in care se pot adauga si sterge elemente la ambele capete, putand fi astfel folosita, printre altele, ca structura FIFO sau LIFO. In acest scop apar metode corespunzatoare celor prezentate in *Queue*, care insa au sufixul First sau Last in functie de capatul la care actioneaza:

	Actioneaza la inceputul cozii		Actioneaza la sfarsitul cozii	
	Arunca exceptie	Returneaza o valoare particulara	Arunca exceptie	Returneaza o valoare particulara
Adaugare element	<code>addFirst(e)</code>	<code>offerFirst(e)</code>	<code>addLast(e)</code>	<code>offerLast(e)</code>
Stergere element	<code>removeFirst()</code>	<code>pollFirst()</code>	<code>removeLast()</code>	<code>pollLast()</code>
Inspectie element	<code>getFirst()</code>	<code>peekFirst()</code>	<code>getLast()</code>	<code>peekLast()</code>

9.1.5.5. Colectii de tip set de corespondente

9.1.5.5.1. Interfata Map

Interfata *Map* modeleaza un grup de corespondente cheie-valoare (ca un tabel cu doua coloane); atat cheia cat si valoarea sunt obiecte. Aceasta reprezinta o generalizare a conceptului de tablou de obiecte, care era un set de corespondente numar intreg→obiect; intr-un *Map* fiecare valoare este referita nu prin intermediul indexului, ci al cheii corespondente. Din acest motiv nu sunt permise cheile dupicat.

O colectie de tip *Map* nu pastreaza ordinea elementelor.

Operatii:

- adaugare/extragere/eliminare de element: **put(Object cheie, Object valoare)**, **get(Object cheie)**, **remove(Object cheie)**
- verificarea existentei unei chei/valori: **containsKey(Object cheie)**, **containsValue(Object valoare)**
- extragerea cheilor sau valorilor sub forma de colectie: **keySet()**, **values()**
- operatii bloc: **putAll(Map)**, **clear()**

9.1.5.5.2. Interfata SortedMap

Reprezinta varianta ordonata pentru *Map*, elementele fiind mentinute in ordinea crescatoare a cheilor (vezi capitolul despre ordonare). Metode suplimentare fata de Map:

- extragerea elementelor ale caror chei sunt mai mici/mai mari decat cheia specificata: **headMap(Object cheie)**, **tailMap(Object cheie)**
- extragerea intervalului de elemente ale caror chei sunt cuprinse intre cheile specificate: **subMap(Object cheie1, Object cheie2)**.

9.1.5.5.3. Interfata NavigableMap

Analog cu NavigableSet, obiectele a caror clasa implementeaza NavigableMap au capabilitati de parcurgere in ambele sensuri a elementelor si de determinare a cheilor mai mici sau mai mari (strict sau nu) decat un obiect dat.

9.1.6. Parcurgerea unei colectii

9.1.6.1. Modalitati

Parcurgerea unei colectii inseamna obtinerea, pe rand, a unei referinte catre fiecare obiect din colectie. Operatia nu ridica probleme in cazul colectiilor care mentin ordinea elementelor (cum este *List*) insa cere masuri suplimentare atunci cand ordinea nu este pastrata, pentru a evita preluarea aceluiasi obiect de doua ori.

Modalitatile de parcurgere a unei colectii sunt urmatoarele, in functie de natura sa:

- pentru colectii de tip *List* se poate folosi bucla *for* clasica
- pentru orice tip de colectie exista alte doua posibilitati:
 - bucla *for* extinsa
 - iteratori

Toate aceste posibilitati vor fi descrise in continuare.

9.1.6.2. Bucla for (pentru colectii de tip List)

Particularitatea colectiilor de tip list este ca ele mentin ordinea elementelor continue si deci ofera conceptul de *pozitie* a unui obiect in cadrul colectiei. Aceasta face ca parcurgerea sa se realizeze facil cu ajutorul unei bucle *for* obisnuite, dotata cu variabila contor pentru pozitie:

```
List colectie = new ArrayList();
// pentru o colectie de obiecte Animal
for(int i=0; i<colectie.size(); i++){
    Animal a = (Animal)colectie.get(i);
    // in continuare poate fi utilizata referinta a
}
```

9.1.6.3. Bucla for-each

Bucla *for...each* a fost introdusa in Java 1.5 si ofera posibilitatea parcurgerii unei colectii chiar si in conditiile in care aceasta nu mentine ordinea elementelor sale:

```
// pentru o colectie obisnuita
Collection colectie = new ArrayList();
// colectia este populata, apoi:
for (Object o : colectie){
    // daca colectia contine obiecte Student:
    Student s = (Student)o;
    System.out.println(o);
}

// pentru o colectie de tip Map
Map m = new HashMap();
// ...iar dupa populare:
for(Object cheie:m.keySet()) { Object valoare = m.get(cheie); ....cod care foloseste valoarea... }
```

De remarcat faptul ca, in acest caz, avem acces secevential la obiectele din colectie; nu ne putem intoarce si nu putem sterge obiectul curent. De asemenea, nu putem itera prin mai multe colectii in paralel folosind aceasta constructie.

9.1.6.4. Iteratori

Iteratorii sunt obiecte ajutatoare care ne ofera posibilitatea de a baleia obiectele dintr-o colectie, chiar si atunci cand in colectia respectiva ordinea nu este garantata (ex: *Set*). Modurile de parcurgere a colectiei depind de conditiile impuse acesteia:

- un Set (sau mai general, Collection) este ca o punga cu obiecte: dam drumul diverselor elemente inauntru, ele pierzandu-si ordinea. Avem posibilitatea de a baleia toate obiectele (introducem mana in punga si scoatem cate unul) cu ajutorul unui iterator, care se asigura ca fiecare obiect este scos o singura data
- intr-o lista (List) fiecare element isi cunoaste predecesorul si succesorul; acestia sunt stabiliti la adaugarea obiectului in lista. Este ca un sir de oameni care se tin de mana: ordinea este stabilita, dar pentru a afla care este al n-lea trebuie parcursa lista de la inceput. Iteratorul ne permite sa parcurgem lista si in sens invers.
- intr-un Map, valorile sunt referite prin intermediul cheilor lor, si nu exista iteratori. In schimb, obiectele de tip Map permit livrarea cheilor sub forma unei colectii, prin care apoi se poate itera, extragand pe rand si valorile corespunzatoare (vezi mai jos)

Un iterator este un obiect care implementeaza interfata *Iterator*. Aceasta nu are decat trei metode:

- **next()** – intoarce urmatorul obiect din colectie; daca colectia este de tip Set (“unga”), nu stim ce obiect va urma (iteratorul nu intoarce obiectele in aceeasi ordine de fiecare data)
- **hasNext()** – verifica daca mai sunt obiecte in “unga”
- **remove()** – elimina din colectie elementul produs de ultima invocare a metodei *next()*

Pentru colectii de tip *List* exista de asemenea interfata derivata *ListIterator*, care adauga metode specifice proprietatii de pastrare a ordinii elementelor: parcurgerea in sens invers a colectiei, inserare, modificare si citire a unui obiect de pe o anumita pozitie etc.

Iteratorul poate fi obtinut cu ajutorul unei metode a colectiei:

- **iterator()** - pentru colectiile non-*List*

- **listIterator()** - pentru cele ce implementeaza *List*

Trebuie subliniat faptul ca iteratorul este un obiect creat pe loc si care reflecta starea colectiei la un moment dat – intre crearea si folosirea sa nu trebuie sa existe modificarile colectiei; in caz contrar iteratorul detecteaza modificarile si arunca o exceptie. Aceasta este si motivul prezentei metodei *remove()* in iterator, desi ea exista si in API-ul colectiei.

Exemplu: data fiind o colectie c, parcurgerea sa poate fi realizata astfel:

```
Iterator i=c.iterator();
while(i.hasNext()) {
    System.out.println(i.next());
}
// implementare alternativa cu for:
for(Iterator i = c.iterator();i.hasNext(); ) {
    System.out.println(i.next());
}
// iterarea in cazul unui Map:
for (Iterator i = m.keySet().iterator();i.hasNext(); ) {
    System.out.println(i.next());
}
```

9.1.7. Operatii cu colectii: clasa Collections

Aceasta clasa contine metode statice ce implementeaza diversi algoritmi optimizati pentru lucrul cu colectii. Cateva exemple:

- **min(Collection)**, **max(Collection)** – returneaza primul/ultimul element din colectie, dupa ordonare (nativa sau cu comparator - vezi sectiunea despre ordonare)
- **shuffle(List)** – amestecarea elementelor listei specificate ca parametru
- **sort(List)** si **sort(List, Comparator)** – ordonarea elementelor unei liste, dupa criteriul implicit sau dupa unul aditional
- **reverseOrder(List)**, **reverseOrder(List,Comparator)** - ordonarea inversa (descendentă) a elementelor colectiei
- **swap(List, int pozitie1, int pozitie2)** – interschimbarea a doua elemente dintr-o lista, specificate prin pozitiile lor in cadrul listei
- **indexOfSubList(List colectieMare, List subcolectie)** - identifica pozitia de inceput a primei aparitii a subcolectiei in cadrul colectiei mari (-1 in caz de absenta)
- **lastIndexOfSubList(List colectieMare, List subcolectie)** - analog cu precedenta, insa identifica pozitia *ultimei* aparitii
- **copy(List destinatie, List sursa)** - suprascrie elementele din *destinatie* cu cele de pe pozitiile corespunzatoare din *sursa*

9.1.8. Ordonarea obiectelor

9.1.8.1. Principii

Ordonarea unei colectii presupune efectuarea de comparatii intre elemente si schimbarea pozitiilor acestora pana cand toate se vor gasi in ordinea dorita, corespunzatoare unui anumit criteriu de ordonare. Pentru a putea ordona obiectele unei colectii trebuie indeplinite doua conditii:

- colectia trebuie sa mentina ordinea elementelor, asadar trebuie sa fie de tip *List*
- trebuie definita o relatie de ordine intre elemente – altfel spus, trebuie indicat dupa ce criterii se efectueaza comparatia intre oricare doua elemente ale colectiei

Nota: putem avea colectii care pastreaza ordinea elementelor dar care nu sunt ordonate (ex: un List inca neordonat), dar si colectii care nu pastreaza ordinea elementelor insa au definita o relatie de ordine! (ex: SortedSet)

Ordonarea unei colectii de tip *List* se realizeaza prin intermediul supraincarcatei *Collections.sort()*:

- **sort(List l)** – ordoneaza colectia primita ca argument dupa criteriul implicit (“native sorting” - vezi mai jos)
- **sort(List l, Comparator criteriu)** – ordoneaza colectia primita ca argument dupa criteriul specificat

Atentie! Ambele metode opereaza direct asupra colectiei primite ca argument, schimbandu-i ordinea elementelor, in loc sa returneze o noua colectie in care obiectele sunt ordonate!

Ordonarea elementelor unei colectii presupune un algoritm cu doua operatii distincte:

- compararea elementelor doua cate doua. Ce elemente se compara si cate comparatii rezulta in final, depinde de algoritmul folosit, insa rezultatul acestor comparatii este cel care va determina ordinea finala a elementelor in colectie. Algoritmul nu poate fi banuit de clasa Collections si trebuie implementat de catre programator
- schimbarea pozitiei elementelor. Atunci cand operatia anterioara decide ca obiectul o1 este "mai mare" decat obiectul o2, insa o2 se afla pe o pozitie anterioara lui o1, trebuie schimbata pozitia obiectelor astfel incat succesiunea lor sa fie cea corecta (procedura exacta depinde din nou de algoritmul folosit). Permutarea elementelor este implementata in clasa Collections

Metoda **sort()** implementeaza cea de-a doua operatie (care nu necesita avizul programatorului), in schimb pentru a o putea realiza pe prima trebuie sa cunoasca dupa ce reguli doreste programatorul sa fie comparate obiectele. Programatorul va scrie o metoda care compara obiectele si aceasta va fi apelata automat de algoritmul de ordonare. Avand in vedere ca regulile de comparare depend de natura obiectului (tipul sau de date), este firesc ca acestea sa nu poata fi parte a metodei **sort()**, ci pot lua doua forme:

- reguli de comparare incluse in chiar clasa obiectului. Pentru ca algoritmul de ordonare sa "fie sigur" de prezenta metodei necesare in clasa respectiva este necesar ca aceasta sa implementeze o interfata – si anume *Comparable*. Aceasta abordare defineste criteriul implicit de ordonare ("native sorting")
- reguli externe, specificate sub forma unui obiect separat. Obiectul implementeaza interfata *Comparator* si este un fel de balanta – primește doua obiecte si returneaza rezultatul compararii lor

9.1.8.2. Ordonarea dupa criteriul default ("native sorting")

Orice obiect care va participa intr-o colectie ordonata dupa criteriul default trebuie sa implementeze interfata *Comparable*; in caz contrar se va arunca o exceptie fie la incercarea de ordonare a colectiei, fie la adaugarea obiectului in colectie in cazul colectiilor care mentin obiectele deja ordonate (ex: *TreeSet*).

Interfata *Comparable* impune o singura metoda claselor ce o implementeaza: **int compareTo(Object o)**. Metoda fiind una de instanta, ea ii va fi apelata unui obiect pentru a-l compara cu cel primit ca argument:

```
o1.compareTo(o2)
```

Rezultatul produs trebuie sa fie:

- negativ, daca o1 este "mai mic" decat o2
- 0, daca o1 este "egal" cu o2 (in sensul metodei equals!)
- pozitiv, daca o1 este "mai mare" decat o2

Date fiind aceste impuneri in privinta rezultatului de iesire, in cazul in care obiectul primit ca argument nu este de tipul de date potrivit (ex: o1 este de tip *Student* iar o2 de tip *Carte*) singura varianta de a semnaliza eroarea este o exceptie. De aceea descrierea metodei (vezi documentatie API) lasa libertatea metodei de a arunca *ClassCastException*, ceea ce permite programatorului sa efectueze conversia argumentului la tipul de date dorit fara alte precautii:

```
class Pisica{
    private String nume;
    public int compareTo(Object o) {
        Pisica p=(Pisica)o;           // Arunca ClassCastException daca o nu e de tipul Pisica
        return nume.compareTo(p.nume); // Compararea de obiecte se reduce aici la
                                       // simpla comparare de nume
    }
}
```

9.1.8.3. Ordonarea dupa criterii aditionale

Procedeul descris mai sus reprezinta *ordonarea nativa* a obiectelor (folosind propria lor metoda de comparare). Ce se intampla insa atunci cand vrem sa efectuam ordonarea si dupa alte criterii, sau pur si simplu obiectele noastre nu implementeaza *Comparable*? Spre exemplu, obiectele de tip *Om* ar putea fi ordonate dupa nume, prenume, telefon, adresa etc., si este foarte probabil ca in cadrul aceleiasi aplicatii sa fie nevoie de toate acestea.

Solutia este folosirea unui obiect special ce realizeaza compararea a doua elemente. Acesta trebuie sa implementeze interfata *Comparator*, ce impune urmatoarea metoda:

```
int compare(Object o1, Object o2)
```

Rezultatul produs respecta aceeasi conventie ca in cazul ordonarii native, insa de data aceasta nu mai suntem constransi la un singur criteriu de ordonare: daca, in cazul lui *Comparable*, clasa ce-l implementa putea avea o *singura* metoda *compareTo()* (si deci un singur criteriu de ordonare a obiectelor), in cazul lui *Comparator* putem crea atatea obiecte *Comparator* cate criterii de ordonare dorim:

```
// ordonarea unei colectii de obiecte Carte dupa titlu si numar de pagini
Comparator dupaTitlu = new Comparator() {
    public int compare(Object o1, Object o2) {
        Carte c1 = (Carte)o1, c2 = (Carte)o2;
        return c1.getTitlu().compareToIgnoreCase(c2.getTitlu());
    }
}

Comparator dupaNrPagini = new Comparator() {
    public int compare(Object o1, Object o2) {
        Carte c1 = (Carte)o1, c2 = (Carte)o2;
        return c1.getNrPagini() - c2.getNrPagini();
    }
}

Collections.sort(carti, dupaTitlu);
Collections.sort(carti, dupaNrPagini);
```

Nota: metodele *Collections.sort()*, *Collections.min()* si *Collections.max()* dispun de o varianta supraincarcata ce primeste ca parametru secund un obiect de tip *Comparator*. De asemenea, colectiile ce-si mentin elementele ordonate (ex: *TreeSet*, *TreeMap*) dispun de o varianta de constructor ce permite specificarea ca argument a comparatorului dorit.

9.1.9. Interfata Enumeration

Interfata *Enumeration* a fost mentionata aici tocmai pentru a o distinge de conceptul de colectie. Ea este implementata de orice obiect ce poate expune utilizatorului sau o lista de elemente - insa interfata nu dispune de metode de adaugare si stergere de elemente, ci permite exclusiv parcurgerea lor.

Interfata *Enumeration* ofera doua metode:

- **hasMoreElements()** – returneaza *boolean*, indicand daca mai sunt elemente neparcurse
- **nextElement()** – returneaza urmatorul element al enumerarii

Printre clasele ce implementeaza interfata *Enumeration* se gasesc:

- clasele colectie mai vechi, create inaintea elaborarii infrastructurii de colectii Java (*Hashtable*, *Vector*), ce dispun de o metoda **elements()** care returneaza un obiect de tip enumerare
- *Iterator*
- *ButtonGroup* (cel folosit pentru gruparea *JRadioButton*-urilor)

Iata un exemplu care afiseaza label-ul *JRadioButton*-ului selectat dintr-un grup:

```
ButtonGroup bg = new ButtonGroup();
[...]
for(Enumeration e = bg.getElements(); e.hasMoreElements(); ) {
    AbstractButton b = e.nextElement();
    if(b.getModel() == bg.getSelected())
        System.out.println(b.getText());
}
```

9.2. Generics: tipuri de date si metode parametrizate

9.2.1. Concepte

O colectie este o constructie de uz general, care isi propune sa-i permita programatorului lucrul cu orice fel de obiecte. Din aceasta cauza metodele sale sunt obligate sa opereze cu cel mai general tip de date: Object. Există însă un efect neplacut: aceasta permite adăugarea în colectie a unui obiect de orice tip derivat din Object - adică, practic, orice obiect Java. În acest fel pot fi create colecții eterogene (care contin, de exemplu, obiecte de tip *Animal*, *Persoana*, *JList*, *Factura* etc). În realitate însă, programatorul folosește colecțiile pentru a depozita obiecte *de același tip* sau de tipuri de date înrudite, la fel ca în cazul tablourilor, și ar dori garanții în privința compozitiei colecției sale.

Acesta este un neajuns surprinzător pentru un limbaj de programare strongly-typed - în care eram obisnuiti ca compilatorul să detecteze orice nepotrivire de tipuri de date în expresii. Aceasta cu atât mai mult că, în cazul tablourilor, verificările *sunt* efectuate: odată declarat tabloul de un anumit tip de date - să zicem, *Animal* - nu vor putea fi adăugate în el decât obiecte de tip *Animal* sau tipuri de date derivate; orice încercare de adăugare a unui obiect de alt tip se va solda cu o eroare de compilare:

```
Animal[] lista = new Animal[2];
lista[0] = new Catzel("Azor");
lista[1] = new Porc("MrProper");
lista[2] = new JButton(); // EROARE DE COMPILEARE!
```

```
List lista = ArrayList();
lista.add(new Catzel("Azor"));
lista.add(new Porc("MrProper"));
lista.add(new JButton()); // FUNCTIEAZA!!
```

Mai constatăm și alte dezavantaje: chiar și în condițiile în care colecția este formată din elemente de același tip, la extragerea unui element compilatorul nu are idee de ce tip este acesta - metoda *get()* a clasei *List*, spre exemplu, returnează *Object* - și suntem astfel obligați la o conversie de referință (cast). Succesul conversiei se bazează pe buna judecata a programatorului; compilatorul nu poate verifica corectitudinea sa, și în consecința evenualelor erori vor fi unele runtime (în spate *ClassCastException*).

```
Animal azor = (Animal)lista.get(0); // arunca eroare runtime în caz de nepotrivire de tip
```

În scopul restabilirii garanțierilor în condițiile lucrului cu colecții (și nu numai), odată cu Java SE 5 s-a introdus facilitatea de *generics* - tipuri de date și metode parametrizate, ce oferă printre altele posibilitatea de a declara o colecție ca acceptând explicit doar un anume tip de obiecte. În acest fel, compilatorul nu va mai permite introducerea unui alt tip de obiecte în colecție, iar la extragerea unui element tipul de date este cunoscut (nu mai este nevoie de cast) și compilatorul poate efectua îăراسi validări, evitându-se erorile tarzii, runtime:

```
List<Animal> lista = new ArrayList<Animal>();
lista.add(new Catzel("Azor"));
lista.add(new Porc("MrProper"));
lista.add(new JButton()); // EROARE DE COMPILEARE
Animal azor = lista.get(0); // nu mai este nevoie de cast
```

În acest exemplu, *List* este un tip de date parametrizat: *List<Animal>* se poate citi ca "lista de obiecte animal" și impune din start compozitia colecției. Din acest moment colecția nu mai poate include obiecte care nu sunt *instanceof Animal*.

9.2.2. Mecanismul de functionare al parametrizarii

Cele de mai sus sunt posibile deoarece unele clase Java (si in special cele colectie) sunt parametrizate: in definitia clasei sau/si a metodelor sale participa un ingredient suplimentar, care este tipul de date impus elementelor colectiei. Iata un fragment din definitia interfetei *List*:

```
public interface List<E> extends Collection<E> {
    boolean add(E e);
    E get(int index);
    .....
}
```

<E> reprezinta un parametru-tip; asa cum o metoda primeste la apelare valori pentru parametrii din definitia ei, in acelasi fel un tip de date parametrizat primeste in momentul folosirii o valoare pentru parametrul-tip. La crearea unui List<Animal>, E este inlocuit cu Animal, iar rezultatul produs de "metoda" List este o definitie particulara de clasa (parte a familiei de clase pe care tipul parametrizat le defineste). Este ca si cum definitia clasei List s-ar transforma in urmatoarea:

```
public interface List extends Collection {
    boolean add(Animal e);
    Animal get(int index);
    .....
}
```

Nota: acest proces are loc fara a multiplica definitia clasei List - exista un singur fisier .class si un singur obiect Class in memorie de care se folosesc compilatorul si masina virtuala, indiferent cate tipuri de colectii List sunt create in program.

Observam ca parametrii-tip permit crearea unei familii de clase/interfete cu definitie comună; odata particularizata definitia clasei/interfetei prin specificarea valorilor parametrilor, compilatorul poate verifica si impune corecta utilizare a tipului de date si a valorilor acestuia.

De acum inainte, acolo unde exista potential de confuzie, vom numi *parametri formali* argumentele primite de catre metode (cele cuprinse intre (...) in definitia metodei) pentru a le distinge de *parametrii-tip* (cei cuprinsi intre <...>).

9.2.3. Sintaxa de definire si utilizare a tipurilor de date si metodelor parametrizate

9.2.3.1. Ce forme iau si unde pot fi utilizati parametrii-tip?

Parametrii-tip sunt utilizabili pentru clase, interfete, metode si constructori. Distingem doua situatii de utilizare:

- la declararea clasei/interfetei/metodei/constructorului. Aici avem *variabile* parametru-tip, ce pot lua urmatoarele forme:
 - **parametri simpli** - de forma <T>
 - **parametri marginiti** - de forma <T extends Clasa> sau <T super Clasa>
- la utilizarea clasei/interfetei/metodei/constructorului, cand variabila parametru-tip trebuie sa capete obligatoriu valoare (la fel ca un parametru formal de metoda). Acest lucru se poate realiza in doua moduri:
 - programatorul precizeaza valoarea pentru parametrul-tip (vezi sectiunile despre utilizarea claselor si metodelor parametrizate). Forme posibile:
 - **valori exacte** - de forma <String>, <Animal> etc
 - **valori de tip wildcard** - de forma <?>
 - **valori de tip wildcard marginite** - de forma <? extends Clasa> sau <? super Clasa>
 - cand valoarea nu este specificata explicit, in anumite cazuri compilatorul o poate deduce din context (procedeul numit *type inference*)

Fiecare dintre aceste sintaxe (plus formele sale aditionale) va fi intalnita/detaliată in sectiunile urmatoare.

9.2.3.2. Tipuri de date parametrizate

9.2.3.2.1. Sintaxa de definire

Atat clasele concrete sau abstracte, cat si interfetele pot fi parametrizate. Acest lucru se realizeaza incluzand in definitia clasei parametri-tip; ei trebuie plasati dupa numele clasei dar inainte de acolada de inceput a definitiei, iar apoi pot fi folositi in interiorul clasei, dupa cum urmeaza:

```
interface Collection<E> extends List<E>{
    boolean add(E element);
    E get(int pozitie);
    .....
}
interface Map<K,V>{
    V put(K key, V value);
    .....
}
```

Numele parametrilor-tip sunt de obicei formate dintr-o singura litera (desi regulile care le guverneaza sunt aceleasi ca in cazul numelor de variabile). Literele folosite in clasele JRE sunt E pentru element, K si V pentru chei si valori, T pentru tip de date si N pentru numar; programatorul are insa libertatea de a alege ce denumiri doreste cand isi creeaza propriile tipuri de date parametrizate.

***Nota:** numim tip de date de baza cel al clasei care a fost parametrizata, si tip de date-parametru cel aflat intre <...> in definitia clasei sau metodei.*

***Atentie!** Clasele de tip exceptie (derivate direct sau indirect din Throwable) nu pot fi parametrizate, deoarece sistemul de prindere a exceptiilor accepta numai clase obisnuite!*

In definitia unui tip de date (clasa/interfata), parametrii-tip pot lua urmatoarele cateva forme:

- parametri simpli, care vor putea lua ca valoare orice tip de date: <T> - indica un tip de date care va primi o valoare in momentul utilizarii clasei/interfetei
- parametri marginiti, al caror domeniu de valori este restrictionat in diverse moduri:
 - <T extends Clasa> sau <T extends Interfata> - restrictioneaza valorile posibile ale parametrului-tip la clasa/interfata specificata si tipurile derivate din ea. *Atentie! Atat pentru clase, cat si pentru interfete se foloseste acelasi cuvant cheie, extends!*
 - <T super Clasa> sau <T super Interfata> - analog, dar permite numai tipuri de date parinte (direct sau indirect) ale clasei/interfetei specificate
 - <T extends Interfata1 & Interfata2 &.....> - impune ca tipul de date primit ca parametru sa implementeze toate interfetele
 - <T extends Clasa & Interfata1 & Interfata2 &.....> - impune ca tipul de date primit ca parametru sa extinda clasa si sa implementeze toate interfetele. **Este permisa o singura clasa si aceea trebuie sa fie prima in cadrul listei!**

9.2.3.2.2. Sintaxa de utilizare - invocarea de tipuri parametrizate

Vom numi referirea la un tip de date parametrizat "invocare". Desi termenul este de obicei rezervat metodelor, realizam ca a folosi un tip de date parametrizat presupune furnizarea de valori pentru parametrii sai tip, ceea ce face operatia asemanatoare cu un apel de metoda.

Invocarea unui tip de date parametrizat se poate produce in doua situatii, guvernate de reguli diferite:

- declararea unei variabile al carei tip de date este parametrizat. Acest caz acopera atat declararea in sine, cat si definitiile de parametri formalii parametrizati pentru metode. In aceasta situatie se permite specificarea de domenii de valori pentru parametrii-tip prin folosirea parametrilor wildcard marginiti:

```
// o lista care va permite numai elemente String
List<String> siruri;
// o lista care va permite orice elemente care sunt instanceof Animal
List<? extends Animal> animale;
// o metoda care primeste ca argument o colectie de animale (dar nu si de tipuri derivate)
public Animal getAnimal(List<Animal> animale, int pozitie){ return animale.get(pozitie); }
// o metoda care primeste ca argument o lista de Animal sau de un anumit subtip de Animal
public <T> T getAnimal(List<? extends Animal> animale, int pozitie){return animale.get(pozitie);}
```

- instantierea unei clase parametrizate. Acest caz este inclus aici deoarece reprezinta tot o forma de invocare a unui tip de date, insa tehnic el tine de utilizarea metodelor, deoarece presupune apelarea constructorului.
- Particularitatea este ca nu mai sunt permisi parametrii de tip wildcard, ci trebuie furnizate valori exacte:

```
// initializarea variabilei siruri de mai devreme
siruri = new ArrayList<String>();
// initializarea variabilei animale;
// elementele colectiei pot fi de un subtip al lui Animal (insa toate de acelasi subtip!)
animal = new LinkedList<Papagal>();
```

9.2.3.3. Metode si constructori parametrizati

9.2.3.3.1. Sintaxa de definire

O metoda parametrizata este una a carei definitie include parametri-tip. O metoda poate fi parametrizata independent de clasa din care face parte: putem avea metode obisnuite in clase parametrizate, metode parametrizate in clase obisnuite sau metode parametrizate in clase parametrizate. In ultimul caz metodele parametrizate pot utiliza parametrii-tip prezenti in definitia clasei (vezi cazul interfetei *List*).

Parametrii-tip pot fi proprii (definiti in cadrul metodei) sau pot proveni din definitia clasei atunci cand aceasta este parametrizata. In functie de acest aspect, definitia metodei trebuie sa includa sau nu o definitie de parametru-tip, dupa cum urmeaza:

- atunci cand metoda foloseste exclusiv parametri-tip definiti in cadrul clasei, ea va fi scrisa ca o metoda obisnuita, numai ca poate folosi pe post de tipuri de date valorile parametrilor-tip ai clasei:

```
interface List<E>{
    public E get(int pozitie);
    public boolean add(E element);
}
class ArrayList<E> implements List<E>{
    // constructor care primeste ca argument o colectie de acelasi tip de elemente
    public ArrayList(Collection<? extends E>){...}
}
```

- atunci cand metoda foloseste parametri-tip proprii (independent de calitatea clasei sale de a fi parametrizata sau nu), acestia trebuie declarati in cadrul definitiei metodei, dupa lista de modificatori/calificatori dar inainte de tipul de date de intoarcere, conform urmatoarei sintaxe:

```
modificator calificatori <ptip1,ptip2,...> tipDateIntoarcere numeMetoda(listaParametriFormali)
```

```
// utilizare de parametri proprii metodei
class Utils {
    public <T> T scoateObiect(List<T> lista, int pozitie){ return lista.get(pozitie); }
    public static <T,U> Map<T,U> makeMap(){ return new HashMap<T,U>(); }
}
// combinatie de parametri-tip proprii si ai clasei
public class Mix<T>{
    public <U> Map<T,U> makeMap(){ return new HashMap<T,U>(); }
}
```

Nota: pentru claritate, precizam ca in implementarile (corful) metodelor din acest exemplu avem mostre de utilizare a metodelor parametrizate (vezi sectiunea urmatoare).

9.2.3.3.2. Sintaxa de utilizare

La apelarea unei metode parametrizate trebuie sa existe o valoare pentru fiecare parametru-tip folosit in cadrul ei.

Distingem urmatoarele situatii:

- valoarea poate fi specificata explicit de catre programator; in cazul metodelor, ea trebuie plasata inaintea numelui metodei, iar in cazul constructorilor intre numele acestuia si lista sa de argumente:

```
Animal a1 = new Animal("Rex"), a2 = new Animal("Fifi");
List<Animal> animale = Arrays.<Animal>asList(new Animal[]{a1,a2});
(new Utils()).<Animal>scoateObiect(animale,1);           // Fifi este cea extrasă
// crearea unei noi liste de animale folosind constructorul care primește ca argument o colecție
List<Animal> altaLista = new ArrayList<Animal>(animale);
```

- valoarea poate lipsi in urmatoarele cazuri:
 - cand metoda foloseste valoarea unui parametru-tip al clasei:

```
List<String> l = new ArrayList<String>();
l.add("Unu"); // metoda add este parametrizată dar folosește numai tipul definit în cadrul clasei
```

- cand valoarea poate fi dedusa de catre compilator (*type inference* - vezi sectiunea urmatoare)

```
List<Animal> l = new ArrayList<>();
// compilatorul deduce valoarea Animal pentru parametrul-tip al lui ArrayList
```

Atentie! Omiterea integrala a valorii parametrului-tip (inclusiv `<>`) in exemplul anterior are un efect diferit! Ea creeaza un asa-numit **raw type** (tip brut); amestecarea tipurilor brute cu cele parametrizate este permisa pentru compatibilitate cu codul neparametrizat pre-Java 5, dar nerecomandata (vezi sectiunea dedicata).

Exista urmatoarea particularitate la apelarea metodelor parametrizate comparativ cu cele obisnuite: atunci cand este specificata explicit valoarea parametrilor-tip nu se mai poate folosi numele scurt al metodei, chiar daca contextul ar fi permis-o; apelul metodei trebuie precedat fie de `this`, fie de numele clasei, in functie de tipul metodei:

```
class Generics{
    public <T> void g(){}
    public void f(){
        <String> g();           // eroare
        this.<String>g();       // corect
    }

    public static <T> void s(){}
    public static void t(){
        <String>s();           // eroare
        Generics.<String>s();  // corect
    }
}
```

9.2.3.4. Deducerea automata a valorii unui parametru-tip (type inference)

In anumite situatii compilatorul poate deduce valoarea unui parametru-tip din contextul in care acesta este utilizat. Facilitatea se aplica in doua situatii:

- la instantierea unui obiect de tip de date parametrizat. Aceasta presupune folosirea asa-numitului *diamond operator* (operatorul romb → `<>`) introdus in Java 7
- la apelarea unei metode parametrizate, ceea ce permite apelarea metodei la fel ca una obisnuita, neparametrizata

In ambele cazuri, pentru orice parametru-tip a carui valoare nu a fost specificata, compilatorul ia, pe rand, urmatoarele masuri:

- analizeaza tipurile de date ale argumentelor. In cazul in care cel putin unul dintre ele foloseste tipul-parametru, compilatorul incerca sa deduca de acolo tipul de date corect
- daca parametrul-tip apare in tipul de date de intoarcere al metodei, compilatorul va analiza valorile returnate de metoda si va incerca deducerea valorii parametrului-tip de acolo
- daca instantierea reprezinta membrul drept al unei atribuiri, compilatorul “trage cu ochiul” in partea stanga si incerca sa deduca pe baza acelui tip de date valorile pentru parametrii-tip nerezolvati
- daca valoarea nu a putut fi dedusa dupa acesti pasi, se va considera automat ca tip de date Object

```
// pentru metoda asList() se deduce valoarea parametrului-tip pe baza argumentelor;
// daca nu existau argumente s-ar fi folosit tipul de date din stanga operatorului de atribuire
Collection<Animal> c = Arrays.asList(new Animal("Rex"), new Animal("Fifi"));
// apelul complet ar fi fost Arrays.<Animal>asList(new Animal("Rex"), new Animal("Fifi"));

// pentru ArrayList-ul de mai jos este dedus tipul Animal pe baza tipului de date din stanga:
List<Animal> l1 = new LinkedList<>(); // <> este diamond operator
// pentru ArrayList este dedusa valoarea parametrului-tip pe baza argumentului din constructor:
List<Animal> l2 = new ArrayList<>(c);

// o metoda care primeste ca argument o colectie de Animal; tipul de date dedus este
// automat Object, deoarece nu mai exista nici argumente, nici valoare stanga care sa ajute
void f(Collection<Animal> c){}
f(new ArrayList<>()); // error: no suitable method found for f(ArrayList<Object>)

// tipurile de date pentru HashMap sunt deduse pe baza tipului de intoarcere al metodei makeMap()
public static <T, U> Map<T, U> makeMap() { return new HashMap<>(); }
```

9.2.4. Problema relatiilor intre tipuri de date parametrizate

Una dintre multele probleme pe care le introduce parametrizarea este cea a determinarii relatiei corecta “is-a” intre tipuri de date parametrizate. Sa privim urmatorul exemplu:

```
Catzel[] haita = new Bichon[10]; // compileaza ok(Bichon este subclasa a lui Catzel)
ArrayList<Catzel> l = new ArrayList<Bichon>(); // NU COMPILEAZA!
```

Intuitia ne spune ca, in virtutea relatiei is-a, un *Bichon* poate fi folosit in locul unui *Catzel* - ceea ce se si intampla in cazul tablourilor. Iata insa ca, pentru tipuri de date parametrizate, intuitia ne inseala: un *ArrayList* de *Bichon* nu este compatibil cu un *ArrayList* de *Catzel*! Confuzia devine si mai mare daca incercam urmatoarea operatie:

```
ArrayList<Catzel> cobai = new ArrayList<>(); // tipul de date dedus de compilator este Catzel
cobai.add(new Bichon()); // FUNCTIONEAZA! deci un Bichon e vazut totusi ca un Catzel!
```

Asadar *putem* folosi un *Bichon* in locul unui *Catzel*! Si atunci cum de nu a functionat atribuirea de mai devreme? Iata un exemplu care evidentiaza unul dintre impiedimente:

```
ArrayList<Catzel> lista = new ArrayList<Bichon>();
lista.add(new Catzel());
```

Daca acest exemplu ar compila, atunci am avea o colectie de bichoni referita cu o variabila de tip colectie de catzel. Insa valoarea parametrului-tip al variabilei *lista* ne-ar permite apelarea metodei *add()* cu argument de tip *Catzel* - si iata cum in colectia de *Bichoni* ar patrunde obiecte de un tip de date interzis! (obiecte care nu sunt neaparat bichoni)

Dar daca tipul de date din stanga ar fi nu parinte, ci subtip pentru cel din dreapta? Iata exemplul:

```
ArrayList<BichonMaltez> bichoni = new ArrayList<Bichon>(); // NU COMPILEAZA!
```

Din nou insucces. Gasim iute o contradictie: referintei *bichoni* i se poate apela, printre altele, metoda *get(int pozitie)*, care conform tipului de date al referintei ar trebui sa returneze *BichonMaltez*. Pe de alta parte, colectia referita este una de *Bichon* si deci metoda ei *get()* ar produce un obiect cu un tip de date parinte (*Bichon*), deci incompatibil!

Concluzie: obiectul parametrizat referit trebuie sa aiba aceeasi valoare a parametrului-tip ca si referinta (nu sunt permise nici subtipuri, nici supertipuri).

Aceeași concluzie se aplică și argumentelor formale parametrizate pasate metodelor, ca în exemplul următor:

```
public Animal getElement(List<Animal> animale, int pozitie){ return animale.get(pozitie); }
List<Bichon> bichoni = new ArrayList<>();
getElement(bichoni); // EROARE! Este ca și cum am scrie List<Animal> animale = bichoni;
```

A se observa însă că upcasting-ul pentru tipul de date de bază funcționează în continuare; concluzia de mai sus se aplică numai pentru tipurile de date parametru:

```
List<Animal> lista = new ArrayList<Animal>(); // compilează; ArrayList implementează List
ArrayList<Animal> lista = new ArrayList<Bichon>(); // nu compilează
```

Având în vedere incompatibilitatea prezentată, este nevoie de o nouă sintaxă care să relaxeze restricțiile impuse de compilator acolo unde este nevoie, indicându-i acestuia că dorim să accepte o întreagă familie de valori pentru un parametru-tip, nu doar o valoare unică; soluția reprezintă wildcard-urile.

9.2.5. Parametri-tip ce folosesc wildcard-uri

Wildcard-urile pot fi folosite la invocarea unui tip de date - nu la definirea și nu la instantiere! - pentru a relaxa restricțiile impuse valorii unui parametru-tip. Există următoarele forme posibile, ca în exemplele de mai jos:

- *<?>* - indică o valoare necunoscută (mai bine spus, neprecizată). În aceste condiții compilatorul va permite orice valoare a parametrului-tip la invocare
- *<? extends Clasa>* sau *<? super Clasa>* - o valoare necunoscută, dar marginita superior; valoarea va trebui să fie subtip al clasei/interfeței
- *<? super Interfata>* - analog, dar valoarea trebuie să fie un tip parinte al celui specificat

```
List<?> colectie = new ArrayList<Animal>();
List<? extends Animal> bichoni = new ArrayList<Bichon>(); // acum merge!
List<? super Bichon> catei = new ArrayList<Catzel>();
List<? extends Comparable> elemente = new ArrayList<String>();
List<? super Set> lista = new ArrayList<Collection>();
```

Nota: *<?>* este echivalent cu *<? extends Object>*.

Wildcard-urile introduc și ele câteva subtilități. Sa privim următorul exemplu:

```
public void adauga(List<? extends Animal> lista, Animal a){
    lista.add(a); // NU COMPILEAZA!!
}
```

Motivul pentru care acest exemplu nu compilează este următorul: metoda primește ca argument o colecție de elemente de un tip de date care, deși margină, este totuși necunoscut (există o întreagă ierarhie de tipuri de date posibile). În aceste condiții compilatorul nu poate permite adăugarea unui obiect *Animal* în colecție, căt timp nu poate să stă dacă, la vremea apelării metodei, colecția primită va fi de tip *animal* sau de un subtip al său!

Concluzie: utilizarea wildcard-ului marginit superior la invocarea unui tip de date colecție face ca în aceasta să nu se mai poată adăuga elemente. În schimb pot fi accesate fără probleme elementele curente ale colecției.

Există totuși posibilitatea de a adăuga elemente într-o colecție invocată cu wildcard-uri, cât timp ii furnizăm compilatorului garanțiile necesare. Dacă în metoda de mai devreme am păsat ca argument o colecție de elemente al căror tip de date este un tip *parinte* al lui *Animal*, compilatorul va avea certitudinea că poate adăuga un *Animal* într-o astfel de listă, deoarece un obiect *Animal* va fi *instanceof* oricare dintre tipurile de date posibile conform wildcard-ului:

```
public void adauga(List<? super Animal> lista, Animal a){
    lista.add(a);           // compileaza ok
}
```

Atenție! Nu uitati ca wildcard-urile se pot folosi numai la invocarea unui tip de date, nu și la definirea sau instantierea lui! Priviti exemplul următor:

```
class Test<? extends Animal>{}                      // gresit!
class Test<T extends Animal>{}                      // corect
List<?> lista = new ArrayList<? super Bichon>();      // gresit!
List<?> lista = new ArrayList<Animal>();              // corect
```

9.2.6. Interoperabilitatea cu cod care nu folosește tipuri parametrizate

Una dintre provocările introducerii parametrizării în Java a fost compatibilitatea cu codul deja scris. Vechile portiuni de cod trebuiau să poată conlucra fără modificări cu noile portiuni de cod care folosesc parametri-tip. Din acest motiv, Java permite încă ignorarea parametrilor-tip și folosirea codului parametrizat ca în variantele pre-Java SE 5, însă cu unele mentiuni și restricții.

Lă invocarea unui tip de date parametrizat fără a specifica valoarea parametrilor-tip se creează un asa-numit tip brut (*raw type*); acesta va fi compatibil cu tipurile parametrizate însă va genera warning-uri (avertizări) la compilare, deoarece compilatorul nu mai poate efectua verificările pe care le faceau posibile tipurile parametrizate:

```
List l = new ArrayList<Animal>();
/* mesaj la compilare:
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
...iar în cazul compilării folosind opțiunea de compilator -Xlint, care oferă mesaje detaliante:
warning: [rawtypes] found raw type: List
        List l = new ArrayList<Animal>();
missing type arguments for generic class List<E>
where E is a type-variable:
    E extends Object declared in interface List
*/

```

Atenție! Un tip de date brut este diferit de toate variantele parametrizate! List nu este totuși nici cu List<?>, nici cu List<> și nici cu List<Object>!

Aceasta combinație de tipuri brute și parametrizate este posibila grație unei facilități a compilatorului Java numită *type erasure*. În momentul în care este compilat cod parametrizat, compilatorul îndeplinește două sarcini, în această ordine:

- verifica validitatea intregului cod folosindu-se de valorile parametrilor-tip
- elibera toata informația legată de parametrii-tip, generând un bytecode unic, "curat", ce funcționează identic cu cel de dinaintea apariției parametrizării. Compilatorul va introduce automat conversii între tipurile de date acolo unde este necesar

Acesta este motivul pentru care, la rulare, mașina virtuală nu mai este consimță de impunerea aplicată elementelor unei colecții parametrizate, și deci aceasta poate conlucra fără impedimente cu una neparametrizată. Singura problema este la compilare, unde compilatorul nu mai poate avea (să oferă) garanții. Parametrizarea este de fapt o modalitate de protecție a codului aplicată la compilare, ea "volatilizându-se" la rulare.

Nota: faceti distinctie intre cod care compileaza fara erori sau warning-uri, cod care compileaza dar emite warning-uri, si cod care nu compileaza! Warning-urile sunt simple avertizari, insa compilarea se incheie cu succes! In functie de codul executat, ele pot produce sau nu erori runtime la rularea programului.

9.3. BIBLIOGRAFIE

- Java Tutorial
 - Colectii
 - <http://docs.oracle.com/javase/tutorial/collections/index.html>
 - Tipuri parametrizate:
 - varianta simplificata: <http://docs.oracle.com/javase/tutorial/java/generics/index.html>
 - varianta detaliata: <http://docs.oracle.com/javase/tutorial/extras/generics/index.html>
- Generics FAQ: <http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>
- Specificatia oficiala a limbajului Java:
 - Parametri-tip: <http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.4>
 - Clase parametrizate: <http://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.1.2>
- Carti:
 - Sun Certified java Programmer 6 Study Guide: http://www.amazon.com/SCJP-Certified-Programmer-Java-310-065/dp/0071591060/ref=sr_1_2?ie=UTF8&qid=1331146452&sr=8-2 - in special capitolul 7: Generics and Collections