METODE AVANSATE DE PROGRAMARE

Conf.univ.dr. Ana Cristina DĂSCĂLESCU

Temtică curs 9

- Mecanismul de serializare a obiectelor
- Colecții de date – Framework Collection

SERIALIZAREA OBIECTELOR

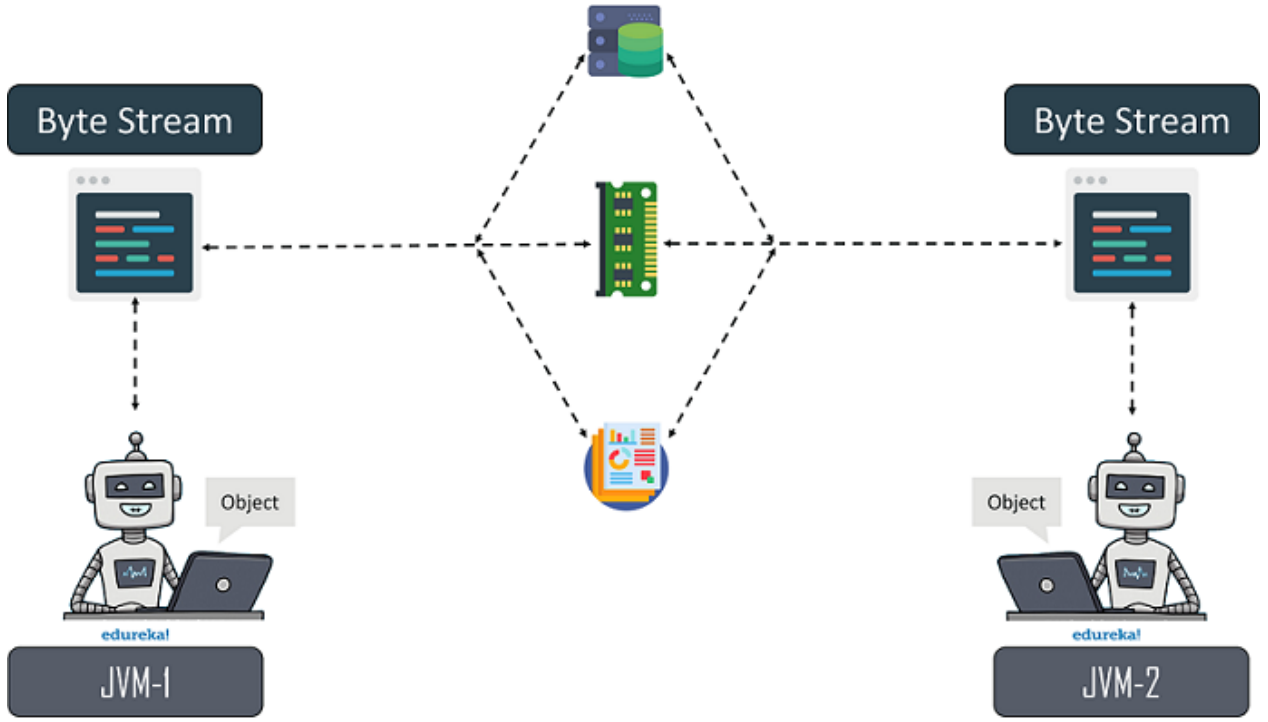
- Salvarea stării unui obiect într-un fișier text sau binar restaurarea ulterioară a acestuia, pe baza valorilor salvate, folosind un constructor al clasei

- Neajunsuri:
  - dacă unele date membre sunt referințe către alte obiecte, atunci ar trebui salvate și restaurate și stările acelor obiecte externe!

  - Nu se pot salva funcționalitățile obiectului (metodele sale) și constructorii

SERIALIZAREA OBIECTELOR

- Limbajul Java permite o rezolvare simplă și eficientă a acestei probleme prin intermediul mecanismelor de **serializare** și **deserializare**



SERIALIZAREA OBIECTELOR

- ***Serializarea*** este mecanismul prin care un obiect este transformat într-o secvență de octeți din care acesta să poată fi refăcut ulterior

- ***Deserializarea*** reprezintă mecanismul invers serializării, respectiv dintr-o secvență de octeți serializați se restaurează obiectul original.

- Utilizarea mecanismului de serializare prezintă mai multe avantaje:
  - obiectele pot fi salvate/restaurate într-un mod unitar pe/de pe diverse medii de stocare (fișiere binare, baze de date etc.);
  - obiectele pot fi transmise foarte simplu între mașini virtuale Java diferite, care pot rula pe calculatoare având arhitecturi sau sisteme de operare diferite;
  - timpul necesar serializării sau deserializării unui obiect este mult mai mic decât timpul necesar salvării sau restaurării unui obiect pe baza valorilor datelor sale member
  - cea mai simplă și mai rapidă metodă de clonare a unui obiect (deep copy) o reprezintă serializare/deserializarea sa într-un/dintr-un tablou de octeți.

SERIALIZAREA OBIECTELOR

- Obiectele unei **clase sunt serializabile dacă respectiva clasă implementează interfața marker `Serializable`.**

- O clasă nu este implicit serializabilă, deoarece clasa **`java.lang.Object`** nu implementează interfața `Serializable`.

- Totuși, anumite clase standard, cum ar fi clasa **`String`**, clasele înfășurătoare (wrapper), clasa `Arrays` etc., implementează interfața `Serializable`.

SERIALIZAREA OBIECTELOR

➢ **Serializarea** unui obiect se realizează astfel:

- se deschide un flux binar de ieșire utilizând clasa **`java.io.ObjectOutputStream`**:

```
FileOutputStream file = new FileOutputStream("studenti.bin");
ObjectOutputStream fout = new ObjectOutputStream(file);
```

- **se salvează/scrie obiectul** în fișier apelând metoda **`void writeObject(Object ob)`**:

```
Student s = new Student("Ion Popescu", 241);
fout.writeObject(s);
```

DESERIALIZARE OBIECTELOR

➢ **Deserializarea unui obiect** se realizează astfel:

- se deschide un flux binar de intrare utilizând clasa **`java.io.ObjectInputStream`**:

```
FileInputStream file = new FileInputStream("studenti.bin");
ObjectInputStream fin = new ObjectInputStream(file);
```

- se citește/restaurează obiectul din fișier apelând metoda **`Object readObject()`**:

```
Student s = (Student)fin.readObject();
```

SERIALIZAREA OBIECTELOR

- Mecanismul de serializare a unui obiect presupune salvarea, în format binar, a următoarelor informații despre acesta:

  - denumirea clasei de apartenență;
  - **versiunea clasei de apartenență, implicit aceasta fiind hash-code-ul acesteia, calculat de către mașina virtuală Java;**
  - valorile datelor membre de instanță.
  - antetele metodelor membre

SERIALIZAREA OBIECTELOR

- Implicit **NU** se serializează datele membre statice și nici corpurile metodelor, ci doar antetele lor.
- Explicit **NU** se serializează datele membre marcate prin modificatorul `transient`
- Serializarea nu tine cont de specificatorii de acces, deci se vor serializa și datele/metodele private!
- **În momentul sterilizării unui obiect se va serializa întregul graf de dependențe asociat obiectului respectiv, adică obiectul respectiv și toate obiectele referite direct sau indirect de el.**

➢Observații

- Dacă un obiect care trebuie serializat conține referințe către obiecte neserializabile, atunci va fi generată o excepție de tipul **`NotSerializableException`**.

- Dacă o clasă serializabilă extinde o clasă neserializabilă, atunci datele membre accesibile ale superclasei nu vor fi serializate.

- Dacă se modifică structura clasei aflată pe mașina virtuală care realizează serializarea obiectelor (ără a se realiza aceeași modificare și pe mașina virtuală destinație, atunci procesul de deserializare va lansa la executare excepția **`InvalidClassException`**

- Versiunea unei clase se poate defini explicit prin data membra

```
private static final long serialVersionUID
```

COLECȚII

- ***Java Collections Framework*** este un framework/API performant pentru crearea și managementul structurilor dinamice de date (tablou, liste, mulțime, tabelă de asocieri etc)

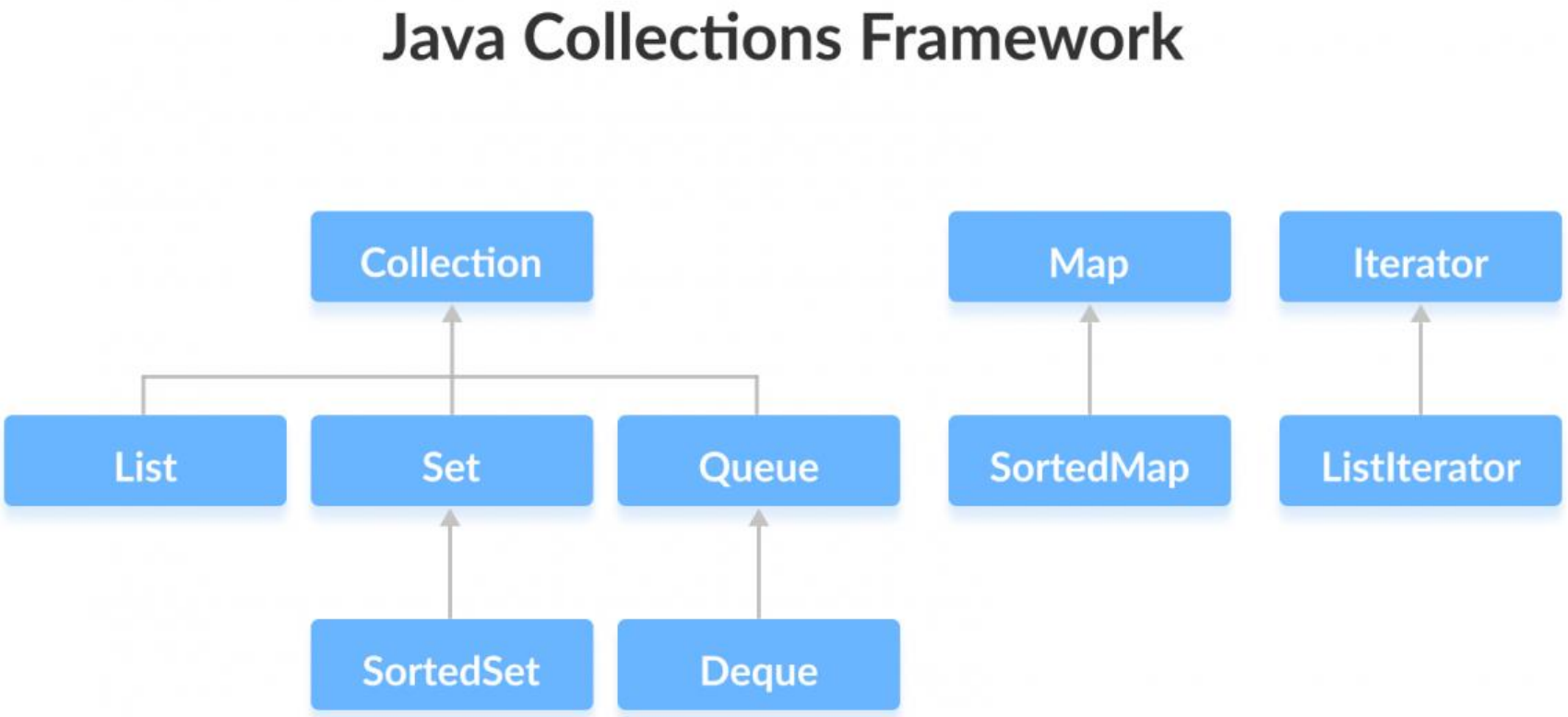
- O **colecție** este un obiect container care grupează mai multe elemente într-o structura unitară de același tip

- **Arhitectura framework-ului Collections**:
  - **Interfețe**: definesc in mod abstract operatiile specifice unei colectii
  - **Clase**: conțin implementări concrete ale colecțiilor definite în interfețe, iar începând cu Java 1.5 ele sunt generice

```
List<Persoana> lp = new ArrayList<>();
```

  - **Algoritmi polimorfici:** metode statice publice, incapsulata intr-o clasa utilitara **Collections**

IERARHIA DE INTERFEȚE



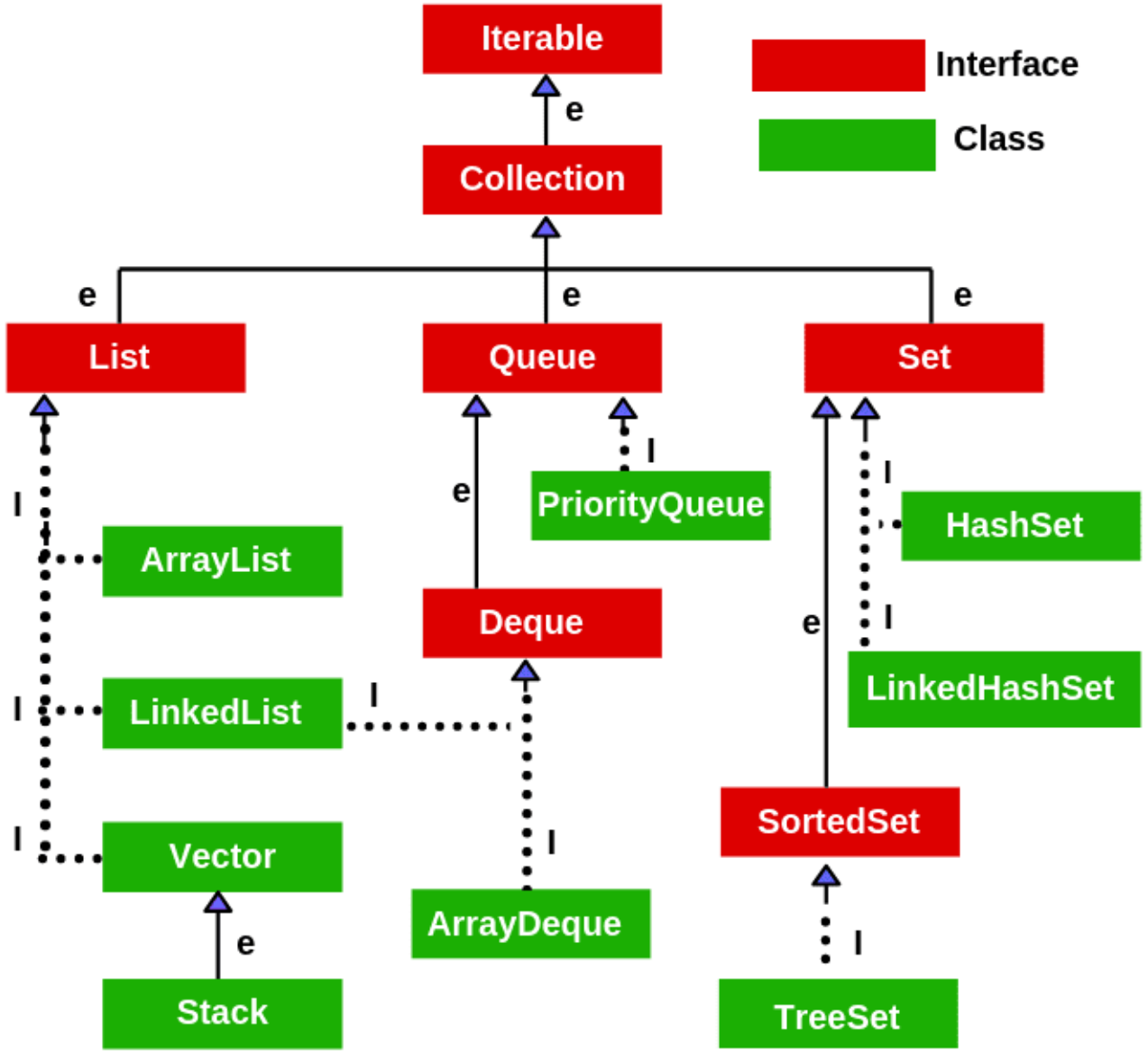
INTERFAȚA COLLECTION

- Conține o serie de metode fundamentale de prelucrare specifice tuturor colecțiilor

- Metodele uzuale ale interfeței `Collection`

- `public int size()` – returnează numărul total de elemente din colecție;
- `public boolean add(E e)` – inserează în colecția curentă elementul `e`;
- `public boolean addAll(Collection<E> c)` – inserează în colecția curentă toate elementele din colecția `c`;
- `public boolean contains(Object e)` – caută în colecția curentă elementul `e`;
- `public Iterator iterator()` – returnează un iterator pentru colecția curentă;
- `public Object[] toArray()` – realizează conversia colecției într-un tablou cu obiecte de tip `Object`.
- `public boolean remove(Object e)` – șterge elementul `e` din colecția curentă;

IERARHIA DE CLASE CARE EXTIND INTERFAȚA COLLECTION



INTERFAȚA LIST

- Extinde interfața `List` și modelează o colecție de elemente ordonate care permite inclusiv memorarea elementelor duplicate.

- Interfața `List` adaugă metode suplimentare fata de interfața `Collection`, corespunzătoare operațiilor care utilizează index-ului fiecărui element, considerat ca fiind de un tip generic `E`

  - accesarea unui element: `E get(int index)`, `E set(int index);`
  - adăugarea/ștergere element: `void add(int index, E element)`, `E remove(int index);`
  - determinarea poziției unui element în cadrul colecției: `int indexOf(Object e)`, `int lastIndexOf(Object e).`

Clasa ArrayList

- Oferă o implementare a unei liste utilizând un **tablou unidimensional** care poate fi redimensionat dinamic

```
List<Tip> listaTablou = new ArrayList<>();
ArrayList<Tip> listaTablou = new ArrayList<>();
```

- Capacitatea implicită a unei astfel de liste este egală cu 10, pentru a specifica explicit o altă capacitate se poate utiliza un constructor care primește ca argument un număr întreg

```
List<Tip> listaTablou = new ArrayList<>(50);
```

➢Observații

- **Accesarea** unui element se realizează cu complexitatea $\mathcal{O}(1)$.
- **Adăugarea** unui element la sfârșitul listei prin metoda **`add(T elem)`** se realizează cu complexitatea $\mathcal{O}(1)$ dacă nu este depășită **capacitatea listei** sau cu complexitatea $\mathcal{O}(n)$ în caz contrar.
- **Adăugarea** unui element pe o anumită poziție prin metoda **`add(E element, int index)`** se realizează cu complexitatea $\mathcal{O}(n)$.
- **Căutarea** sau **ștergerea** unui element se realizează cu complexitatea $\mathcal{O}(n)$.

Clasa LinkedList

- Oferă o implementare a unei liste utilizând o **listă dublu înlănțuită**, astfel fiecare nod al listei conține o informație de tip generic `E`, precum și două referințe: una către nodul anterior și una către nodul următor.

- Constructorii clasei `LinkedList` sunt:
  - `LinkedList()` – creează o listă vidă;
  - `LinkedList(Collection C)` – creează o listă din elementele colecției `C`.

- Pe lângă metodele implementate din interfața List, clasa LinkedList conține și câteva metode specifice:

  - accesarea primului/ultimului element: `E getFirst(), E getLast();`
  - adăugarea la începutul/sfârșitul listei: `void addFirst(E elem), void addLast(E elem);`

Clasa LinkedList

➢ Observații

- **Accesarea** unui element se realizează cu complexitatea $\mathcal{O}(n)$.
- **Adăugarea** unui element la sfârșitul listei, folosind metoda **`add(E elem)`**, se realizează cu complexitatea $\mathcal{O}(1)$
- **Adăugarea** unui element pe poziția **`index`**, folosind metoda **`add(E elem, int index)`**, se realizează cu o complexitate egală cu $\mathcal{O}(n)$.
- Căutarea unui element se realizează cu o complexitate egală cu $\mathcal{O}(n)$.
- Ștergerea unui element se realizează cu o complexitate egală cu $\mathcal{O}(1)$, dacă se specifică indexul elementului.

Interfața Set

- Interfața **`Set`** extinde interfața **`Collection`** și modelează o colecție de elemente care nu conțin duplicate, respectiv o colecție de tip mulțime.

- Interfața **`Set`** nu adaugă metode suplimentare celor existente în interfața **`List`** și este implementată în clasele

✓ `HashSet`

✓ `TreeSet`

✓ `LinkedHashSet`

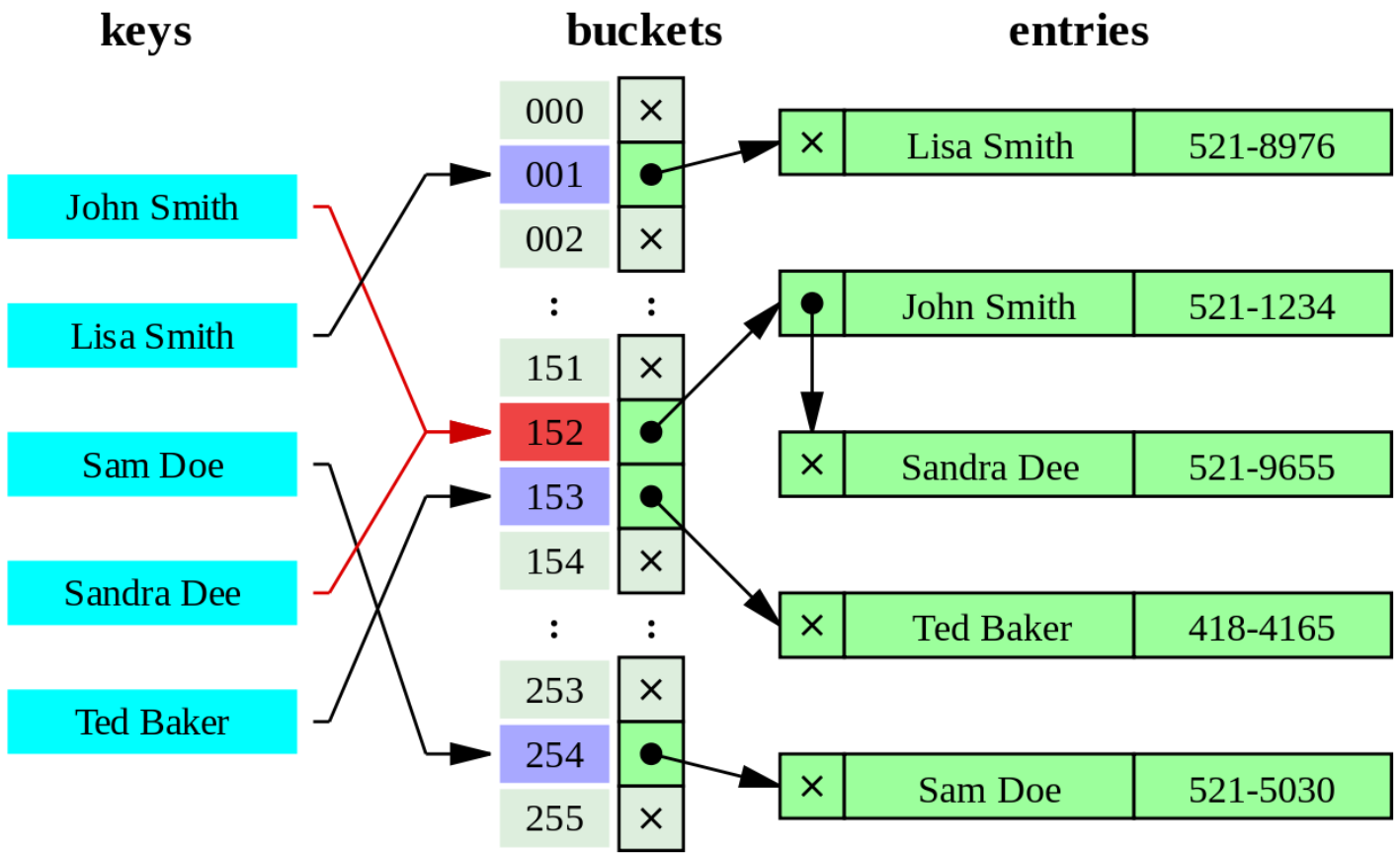
Clasa HashSet

- Clasa **`HashSet`** implementează o mulțime folosind o tabelă de dispersie (hash table)

- O tabelă de dispersie este un **tablou unidimensional**, numit ***bucket array***, în care indexul unui element se calculează pe baza **hash-code**-ului său

➢ Fiecare componentă a bucket array-ului va conține o listă cu obiectele care au același hash-code (*coliziuni*),

Clasa HashSet



Clasa HashSet

- O operație de inserare a unui obiect în tabela de dispersie presupune parcurgerea următorilor pași:

  - se apelează metoda **`hashCode`** a obiectului respectiv, iar valoarea obținută se utilizează pentru a calcula **indexul bucket-ului asociat** obiectului respectiv;

  - dacă bucket-ul respectiv este vid, atunci se adaugă direct obiectul respectiv și operația de inserare se încheie;

  - dacă bucket-ul respectiv nu este vid, se parcurge lista asociată și, folosind metoda **equals**, se verifică dacă obiectul este deja inserat în tabelă, iar în caz negativ obiectul se adaugă la sfârșitul listei.

Clasa HashSet

➢ Observații

- Performanțele unei tabele de dispersie sunt puternic influențate de performanțele algoritmului de calcul al hash-code-ului unui obiect!!!
- Dacă nu există coliziuni, atunci operațiile de **căutare/inserare/ștergere/modificare** vor avea complexitatea $\boldsymbol{\mathcal{O}(1)}$
- O colecție de tip `HashSet` nu păstrează elementele în ordine inserării lor și nici nu pot efectua operații de sortare asupra sa.

Clasa HashSet

➢ Observații

- Într-un `HashSet` se poate insera și valoare `null`, evident, o singură dată.

- Implicit, capacitatea inițială (numărul de bucket-uri) a unei colecții de tip HashSet este 16, iar apoi aceasta este incrementată pe măsură ce se inserează elemente.

- Este definit un **factor de umplere (load factor)** care reprezintă pragul maxim permis de populare a colecției, depășirea sa conducând la dublarea capacității acesteia.
- Implicit, factorul de umplere este egal cu valoarea 0.75, de bucket-uri

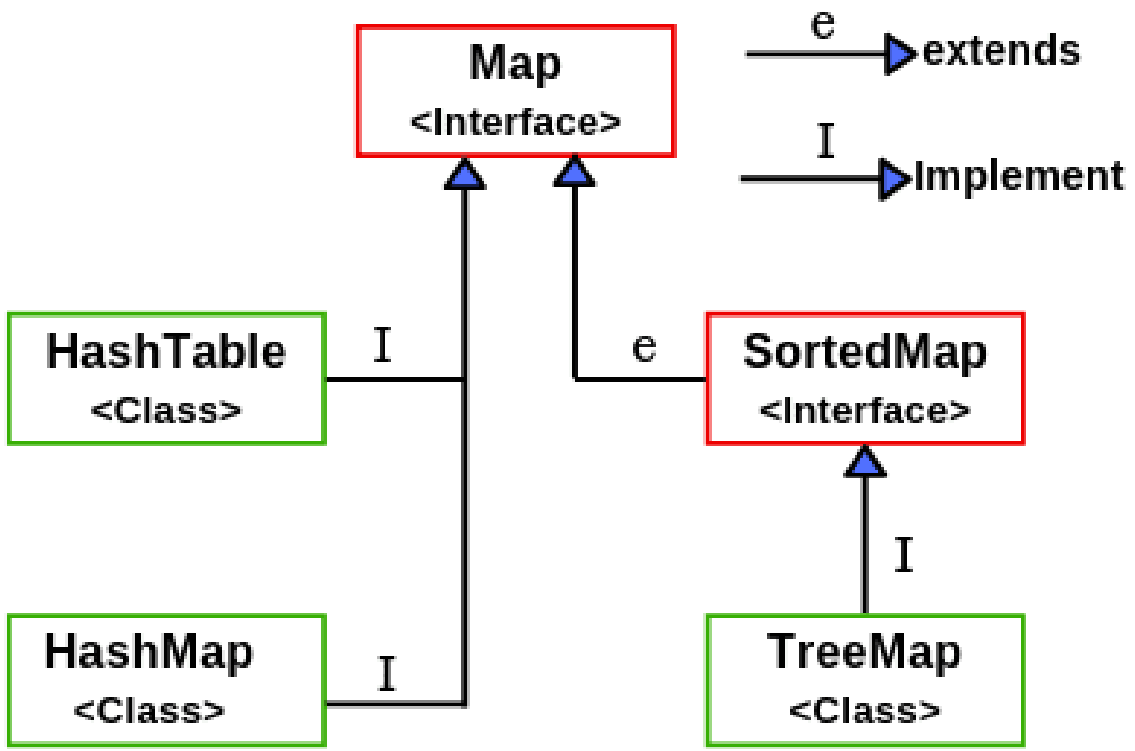
Clasele LinkedHashSet și TreeSet

- Implementarea clasei **`LinkedHashSet`** este similară cu implementarea clasei **`HashSet`**, diferența constând în faptul că elementele vor fi stocate în ordinea inserării lor!!!

- Clasa TreeSet implementează o mulțime utilizând un **arbore binar de tip Red-Black** pentru a stoca elemente într-o anumită ordine.
  - în ordinea lor naturală când se utilizează constructorul fără parametri ai clasei și clasa corespunzătoare obiectelor implementează interfața **Comparable**
  - într-o ordine specificată în constructorul clasei printr-un argument de tip **Comparator**

Interfața Map

- Interfața **`Map`** modelează comportamentul colecțiilor ale căror elemente sunt de perechi de tipul cheie – valoare (definite în interfața **`Map.Entry<T,R>)`**, prin care se asociază unei chei care trebuie să fie unice o singură valoare.



Clasele LinkedHashSet și TreeSet

➢ Metode uzuale din interfața Map

- `R put (T cheie, R valoare)` – inserează perechea cheie-valoare în colecție în cazul în care cheia nu există deja
- `R get(T cheie)` – returnează valoarea asociată cheii indicate sau null dacă în colecție nu există cheia respectivă;
- `boolean containsKey(T cheie)` – returnează true dacă în colecție există cheia respectivă sau false în caz contrar;
- `boolean containsValue(R valoare)` – returnează true dacă în colecție există valoarea respectivă sau false în caz contrar;

Clasele LinkedHashSet și TreeSet

➢ Metode uzuale din interfața Map

- `Set<Map.Entry<K,V>> entrySet()` – returnează o mulțime care conține toate perechile cheie-valoare din colecție;
- `Set<K> keySet()` – returnează o mulțime care conține toate cheile din colecție;
- `Collection<V> values()` – returnează o colecție care conține toate valorile din colecția de tip Map;
- `R remove(Object cheie)` – dacă în colecție există cheia indicată, atunci elimină din colecție perechea având cheia respectivă și returnează valoarea cu care era asociată, altfel returnează null

Implementările Interfeței Map

- Implementarea clasei **`HashMap`** utilizează o tabelă de dispersie în care indexul bucket-ului dat de hash-code-ul corespunzător cheii (**cheie.hashCode()**)

- Complexitățile minime și medii ale metodelor `get`, `put`, `containsKey` și `remove` sunt $\mathcal{O}(1)$ în cazul implementării în metoda hashCode() a unei funcții de dispersie bune

- Într-un **`HashMap`** nu se menține ordinea de inserare și nici nu se poate stabili o anumită ordine a perechilor!

- Într-un **`HashMap`** se poate realiza și o mapare cu o altă colecție care poate fi de orice tip!!!

Implementările Interfeței Map

- Implementarea clasei **`TreeMap`** utilizează un arbore binar de tip **Red-Black** pentru a menține perechile cheie-valoare sortate fie

  - în ordine naturală a cheilor, dacă se utilizează constructorul fără parametri

  - în ordinea indusă de un comparator transmis ca parametru al constructorului

  - Perechile dintr-un **`TreeMap`** pot fi sortate folosind criterii care implică doar cheile

  - Operațiile de inserare/căutare/ștergere într-un **`TreeMap`** se realizează tot pe baza hash-code-ului corespunzător cheii, dar utilizarea unui arbore Red-Black garantează o complexitate egală cu $\mathcal{O}(\log_2 n)$ pentru metodele **`get`**, **`put`**, **`containsKey`** și **`remove`**

Interfața ITERATOR

- Rolul general al unui iterator este acela de a parcurge elementele unei colecții de orice tip, **mai puțin a celor care fac parte din ierarhia interfeței Map**.

- Orice colecție din ierarhia interfeței Collection conține o implementare a metodei **`iterator()`** care returnează un obiect de tip **`Iterator<Tip>:`**

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

- Metode pentru accesarea elementelor unei colecții:

✓ `public Object next()` – returnează succesorul elementului curent;

✓ `public boolean hasNext()` – returnează true dacă în colecție mai există elemente nevizitate sau false în caz contrar.

- Un iterator nu permite modificarea valorii elementului curent și nici adăugarea unor elemente noi în colecție!

Interfața ITERATOR

➢ Fail fast iterator

```
List<Integer> numere = new ArrayList<Integer>();

numere.add(101);
.................

Iterator<Integer> itr = numere.iterator();
while (itr.hasNext()) {
  Integer nr = itr.next();
  if (nr % 2 == 0)
      numere.remove(nr);
}
```

ConcurrentModificationException

Soutile:

```
      itr.remove();
```

Clasa Collections

- Clasa **java.util.Collections** este o componentă importantă a framework-ului `Collection` care oferă implementări specifice colectiilor de date, precum sortare, calcul minim/maxim, căutare, algoritmi de umplere etc.

- Metodele încapsulate în clasa `Collections` fie prelucrează o colecție de elemente, fie returnează un anumit tip de colecție și toate "aruncă" excepția **NullPointerException**

- **Metode uzuale:**

  - **static boolean addAll(Collection<?> c, T... elements)**
  - **static void sort(List<?> list)**
  - **static void sort(List<?> list, Comparator c)**
    - ✓ **Collections.sort(listaIntregi, Comparator.reverseOrder());**
  - **static void copy(List<?> dest, List<?> src**