ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

ΚΕΝΤΡΟ ΕΠΙΜΟΡΦΩΣΗΣ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗΣ

Java Collections

Αθ. Ανδρούτσος

Δομές Δεδομένων (1)

- Λέγοντας **Δομές Δεδομένων** εννοούμε την οργάνωση στη μνήμη συλλογών δεδομένων, (**collection**) οποιουδήποτε τύπου
- Για παράδειγμα οι πίνακες της Java είναι μία γραμμική δομή δεδομένων. Γραμμική σημαίνει ότι υπάρχει σχέση προηγούμενος-επόμενος
- Επίσης, **οι πίνακες στη Java είναι μία στατική δομή δεδομένων**, δηλαδή έχουν σταθερό μέγεθος που δεν αλλάζει δυναμικά (δεν αυξάνεται, δεν μειώνεται, non-modifiable)

Δομές Δεδομένων (2)

- Η **Java μας παρέχει έτοιμες δομές δεδομένων** και έτσι δεν είναι απαραίτητο να δημιουργούμε τις δικές μας δομές δεδομένων, όπως κάναμε με τον διπλά συνδεδεμένη λίστα του παραδείγματος του προηγούμενου κεφαλαίου
- Σε εμπορικές εφαρμογές, μπορούμε απλά να χρησιμοποιούμε τις έτοιμες δομές δεδομένων της Java, που ονομάζονται **Java Collections**

Δομές Δεδομένων (3)

- Οι δομές δεδομένων είναι **κατά βάση συλλογές στοιχείων (collections)** και μπορούμε να κάνουμε σε αυτές CRUD πράξεις
- Ανάλογα με τις ανάγκες τις εφαρμογής μας και τα τεχνικά χαρακτηριστικά κάθε έτοιμης δομής δεδομένων (π.χ. την πολυπλοκότητα χρόνου κάθε πράξης) μπορούμε να επιλέξουμε ποια δομή δεδομένων μας εξυπηρετεί καλύτερα για να χρησιμοποιήσουμε

Δομές Δεδομένων (4)

- Για παράδειγμα ο χρόνος αναζήτησης σε όλες τις γραμμικές δομές είναι γραμμικός O(n) εκτός από το HashMap που είναι σταθερός O(1)
- Υπάρχουν και άλλα κριτήρια επιλογής, όπως αν η δομή είναι στατική ή δυναμική ή αν επιτρέπει διπλότυπα ή όχι
- Για παράδειγμα οι πίνακες είναι στατική δομή, η ArrayList (που έχουμε δει) είναι δυναμική δομή. Η ArrayList επιτρέπει διπλότυπα, τα Sets δεν επιτρέπουν διπλότυπα

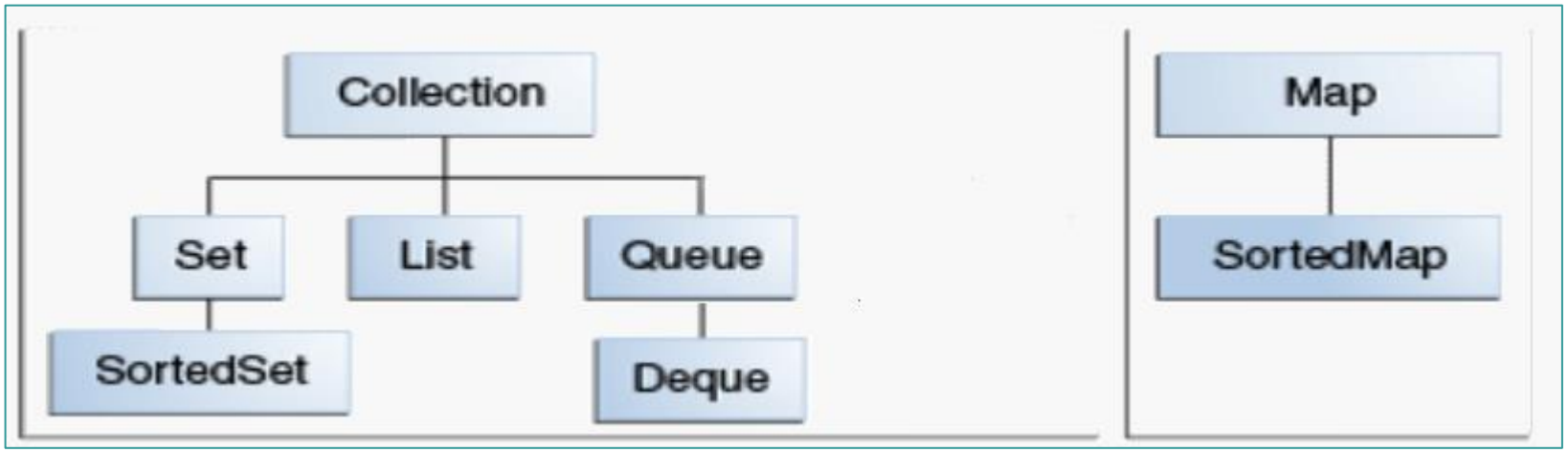
Δομές Δεδομένων (5)

- Αν λοιπόν η αναζήτηση είναι η πιο συχνή πράξη της εφαρμογής μας θα θέλαμε **HashMap**
- Αν θέλουμε και αναζήτηση και όχι διπλότυπα, υπάρχει το **HashSet**
- Αν θέλουμε δομές LIFO, FIFO υπάρχει το interface **Deque** που υλοποιείται από τις κλάσεις **LinkedList** (δυναμική δομή Doubly LinkedList) και **ArrayDeque** (στατική δομή που υλοποιείται με πίνακα)

Java Collections

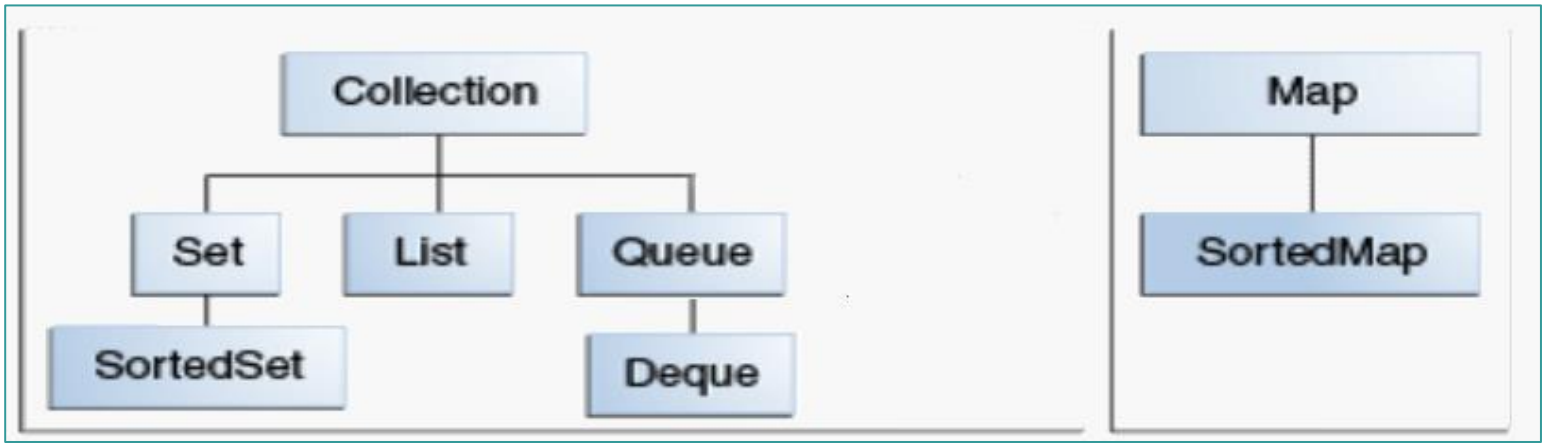
- Το **Java Collections Framework** παρέχει ένα σύνολο από interfaces, κλάσεις που υλοποιούν τα interfaces καθώς και αλγόριθμους που μας επιτρέπουν να κάνουμε CRUD πράξεις (insert, update, remove, search) καθώς και άλλες πράξεις όπως sorting
- Τα Java Collections μειώνουν τον προγραμματιστικό χρόνο ανάπτυξης εφαρμογών μέσω της δυνατότητας επαναχρησιμοποίησης των έτοιμων αυτών δομών δεδομένων

Core Interfaces (1)



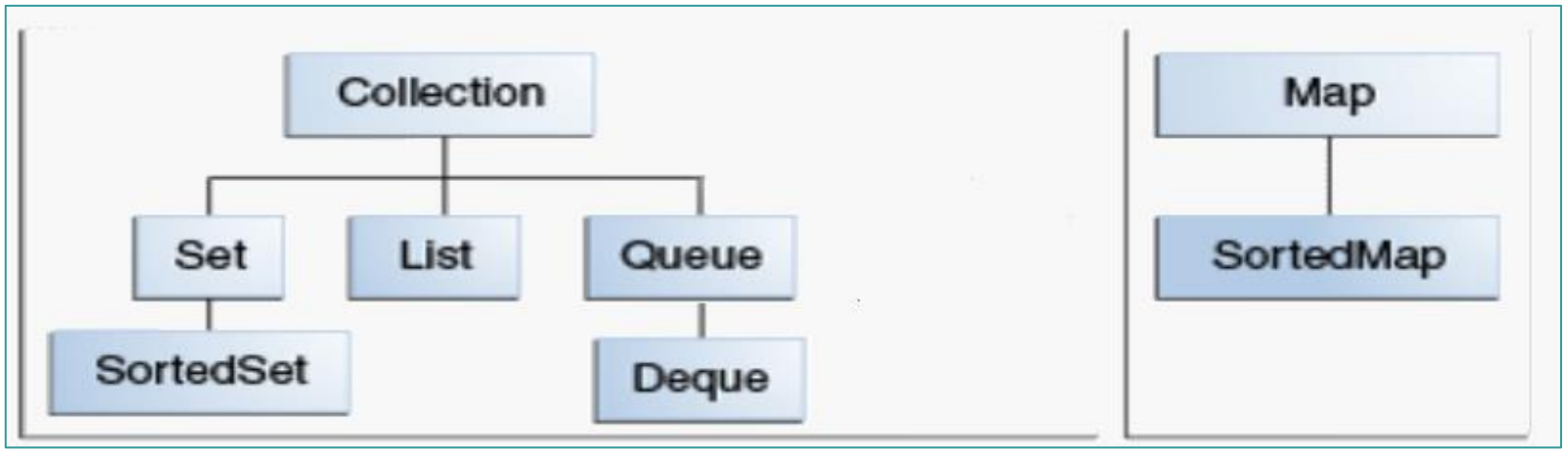
- Ψηλότερα στην ιεραρχία των collection είναι το interface **Collection**, που είναι **Iterable** δηλαδή μπορούμε να διασχίσουμε οτιδήποτε is-a Collection με for-each

Core Interfaces (2)



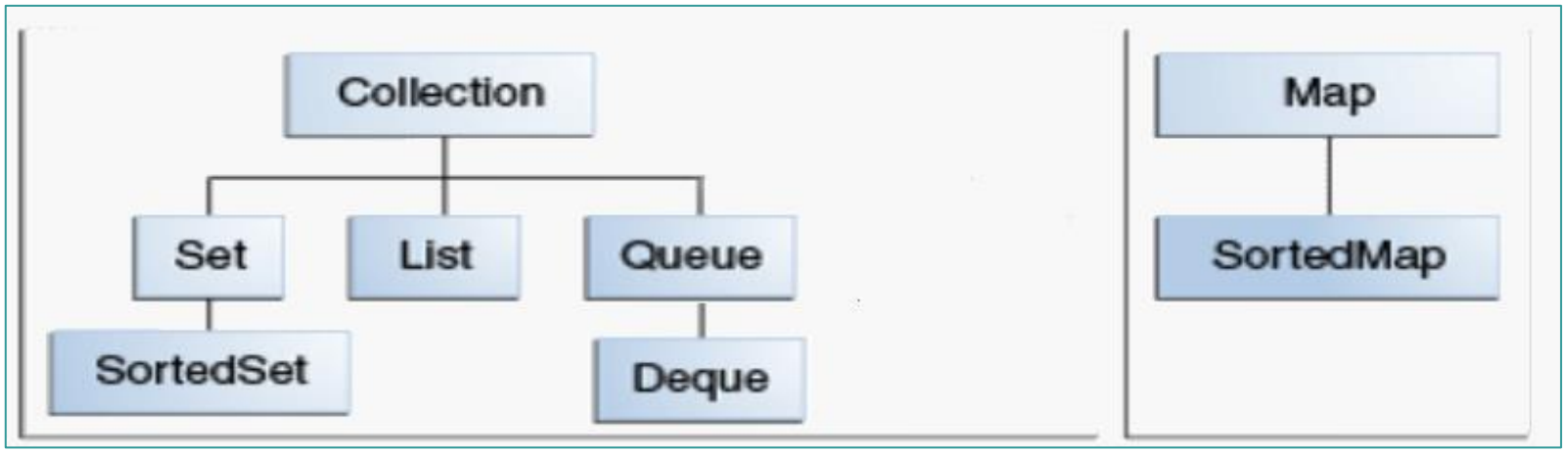
- **Set**. Ένα collection **που δεν επιτρέπει διπλότυπα** (για παράδειγμα τα χαρτιά μιας τράπουλας, τα μαθήματα ενός μαθητή)
- **List**. Πρόκειται για **ordered collection**, μία ακολουθία (sequence) στοιχείων. Τα στοιχεία γίνονται access με τον index (position στην λίστα)
- **Queue**. Μία δομή που υλοποιεί δομές FIFO (ουρά τραπέζης, ουρά επεξεργαστή, ουρά αυτοκινήτων σε ένα φανάρι)

Core Interfaces (3)



- **Deque** (Double-ended queue). Μπορεί να χρησιμοποιηθεί για FIFO/LIFO
- **SortedSet**. Ένα Set που διατηρεί τα στοιχεία του sorted (by default σε αύξουσα σειρά (Asc) αλλά μπορεί να αλλάξει). Υλοποιείται από την κλάση TreeSet<T>

Core Interfaces (4)



- **Map**. Αντιστοιχεί keys σε values. Δεν επιτρέπει διπλότυπα keys
- **SortedMap**. Ένα Map που διατηρεί τα στοιχεία του ordered (by default σε αύξουσα σειρά (Asc) αλλά μπορεί να αλλάξει). Υλοποιείται από την κλάση TreeMap<K, V>

Core Interfaces (5)

- Το Collection interface εξασφαλίζει generality μεταξύ των collections
- Για παράδειγμα, όλα **τα classes που υλοποιούν το interface Collection παρέχουν ένα Constructor που παίρνει ένα Collection argument και επομένως επιτρέπει να κάνουμε *convert* ένα collection type σε ένα άλλο**

Interface Collection

- Το Collection interface παρέχει μεθόδους όπως int **size()**, boolean **isEmpty()**, boolean **add(E element)**, boolean **remove(E element)** και Iterator<E> **iterator()**
- Επίσης παρέχει μεθόδους που επενεργούν επί ολόκληρων collections (bulk operations), όπως boolean addAll(Collection<? Extends E> c), boolean removeAll(Collection<?> c), boolean containsAll(Collection<?> c), και void clear()
- Επιπλέον μέθοδοι παρέχονται για μετατροπή σε arrays όπως Object[] **toArray()** και <T> T[] **toArray(T[] a)**
- Στην JDK8 και μετά παρέχεται η μέθοδος **Stream<E> stream()** και Stream<E> **parallelStream()** που παρέχει sequential ή parallel streams μίας συλλογής

Traverse Collections

- Διάσχιση ενός collection στην Java κάνουμε με

1. **Enhanced For – each** π.χ. for (Object o : collection) {} . Επίσης μπορούμε να διασχίσουμε με **.forEach()**

2. **Iterators**. Ο iterator είναι ένα object που λειτουργεί ως δείκτης στα στοιχεία του collection και μπορεί να κάνει selectively removes. Μπορούμε να πάρουμε ένα iterator από ένα collection με την μέθοδο iterator(). Η *Iterator.remove()* είναι ο μόνος safe τρόπος να κάνουμε modify ένα collection κατά τη διάρκεια του iteration

Filtering, Mapping

- Στο JDK8 και μετά ο προτιμότερος τρόπος για filtering / mapping / sorting ενός collection είναι με **Aggregate operations μέσω της stream()**, όπως filter, map, κλπ.
- Το πλεονέκτημα των aggregate methods είναι ότι παρέχουν ένα functional paradigm για την επεξεργασία collections και δεν κάνουν modify το underlined collection αλλά επιστρέφουν νέο collection

ArrayList (1)

- Η **ArrayList** λόγω της απλότητάς της είναι η πιο συχνά χρησιμοποιούμενη δομή της Java
- Έχει σταθερό χρόνο εισαγωγής στο τέλος, γραμμικό χρόνο εισαγωγής στην αρχή (γιατί μετακινεί όλα τα στοιχεία μία θέση προς το τέλος) και γραμμικό χρόνο αναζήτησης και διαγραφής.
- Συνίσταται να χρησιμοποιείται όταν δεν έχουμε πολλά στοιχεία στη συλλογή μας γιατί το γραμμικό κόστος πολυπλοκότητας χρόνου ιδιαίτερα στις αναζητήσεις δεν είναι αποδοτικό

ArrayList (2)

- Η ***java.util.ArrayList*** *υ*λοποιείται με τη δομή του απλού πίνακα, και όπως ο πίνακας, έχει index από 0 μέχρι size()-1.
- Ωστόσο αυξάνει το μέγεθός της αυτόματα κάθε φορά που γεμίζει, δηλαδή δημιουργεί ένα νέο πίνακα με μέγεθος (*3/2+1) και αντιγράφει τα στοιχεία στο νέο πίνακα
- Πρόκειται για γραμμική δομή Τα στοιχεία είναι διατεταγμένα με αρχική τιμή initial capacity 10 θέσεων
- Δήλωση με Generics, ***ArrayList<T>*** όπου *T* είναι μία κλάση της Java όπως Integer, Float, Double, Boolean, String, κλπ. ή μία user-defined κλάση
- Η Java δεν επιτρέπει primitives στα Generics

ArrayList / LinkedList / HashMap

- Αν θέλουμε γραμμική δομή και έχουμε πολλές εισαγωγές στην αρχή είναι καλύτερη η *LinkedList* που υλοποιείται με δυναμική λίστα διπλής κατεύθυνσης (όπως η Doubly Linked List που υλοποιήσαμε!)
- Αν έχουμε πολλές αναζητήσεις καλύτερα να χρησιμοποιούμε Maps όπως *HashMap* που όμως δεν έχει σειρά ταξινόμησης

List / AbstractList / ArrayList

- Η δήλωση της *ArrayList* στην Java ακολουθεί το μοντέλο που έχουμε δει, δηλαδή κληρονομεί από την *AbstractList* η οποία υλοποιεί το *interface List*
- Επομένως μπορούμε να κάνουμε δηλώσεις της μορφής:

List<E> myList = new ArrayList<>();

List - ArrayList

- Στην προηγούμενη δήλωση, *List<E> myList = new ArrayList<>();* στο δεξί μέρος το **<>** είναι κενό αλλά υπονοείται *<E>*
- Επίσης, η συγκεκριμένη μορφή δήλωσης υπερτερεί της *ArrayList<E> myList = new ArrayList<>();* γιατί στην 1η περίπτωση μπορούμε να χρησιμοποιήσουμε ως myList οποιαδήποτε List όπως LinkedList αλλάζοντας σε *List<E> myList = new LiknedList<>();* με διαφάνεια προς τον χρήστη

Χρήσιμες μέθοδοι της ArrayList

- **boolean add(e)** – ***εισάγει*** στο τέλος (της λίστας) το **element e**
- **void add(index, e) – εισάγει το e στη θέση index** και μετακινεί δεξιά όλα τα άλλα στοιχεία να υπάρχουν
- **E set(index, e)** – ***ενημερώνει*** τη θέση index (της λίστας) με νέο στοιχείο e. Επιστρέφει το προηγούμενο στοιχείο
- **item get(index)** – ***επιστρέφει το στοιχείο*** **item** από τη θέση **index** της λίστας
- **E remove(index)** – ***διαγράφει*** (το στοιχείο **item**) από τη θέση **index.** Επιστρέφει το διαγραμμένο στοιχείο
- **boolean remove(object)** – ***διαγράφει*** (το αντικείμενο object)
- **Boolean removeIf(Predicate<? Super E>)**
- **int indexOf(e)** – ***επιστρέφει τη θέση*** (στη λίστα) του στοιχείου **e**

equals

- Οι int indexOf(e) και boolean remove(obj) ελέγχουν την ισότητα του e με τα αντικείμενα της λίστας και επομένως χρησιμοποιούν την equals
- Για να δουλέψει σωστά το παραπάνω θα πρέπει η equals στις κλάσεις που περνάμε στην ArrayList να έχουμε υπερκαλύψει την equals (και hashCode)

Collections (1)

- Η Java όπως αναφέραμε παρέχει το interface ***java.util.Collection*** που ορίζει τις βασικές πράξεις (Pubic API) σε ομάδες ομοειδών αντικειμένων δηλαδή **συλλογών**
- Το interface **Collection** βρίσκεται στην κορυφή της ιεραρχίας των collections της Java

Interface Collection

Η Java ορίζει τρία βασικά sub-interfaces του Collection
Πάνω σε αυτά γίνονται οι υλοποιήσεις των concrete κλάσεων

docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collection.html

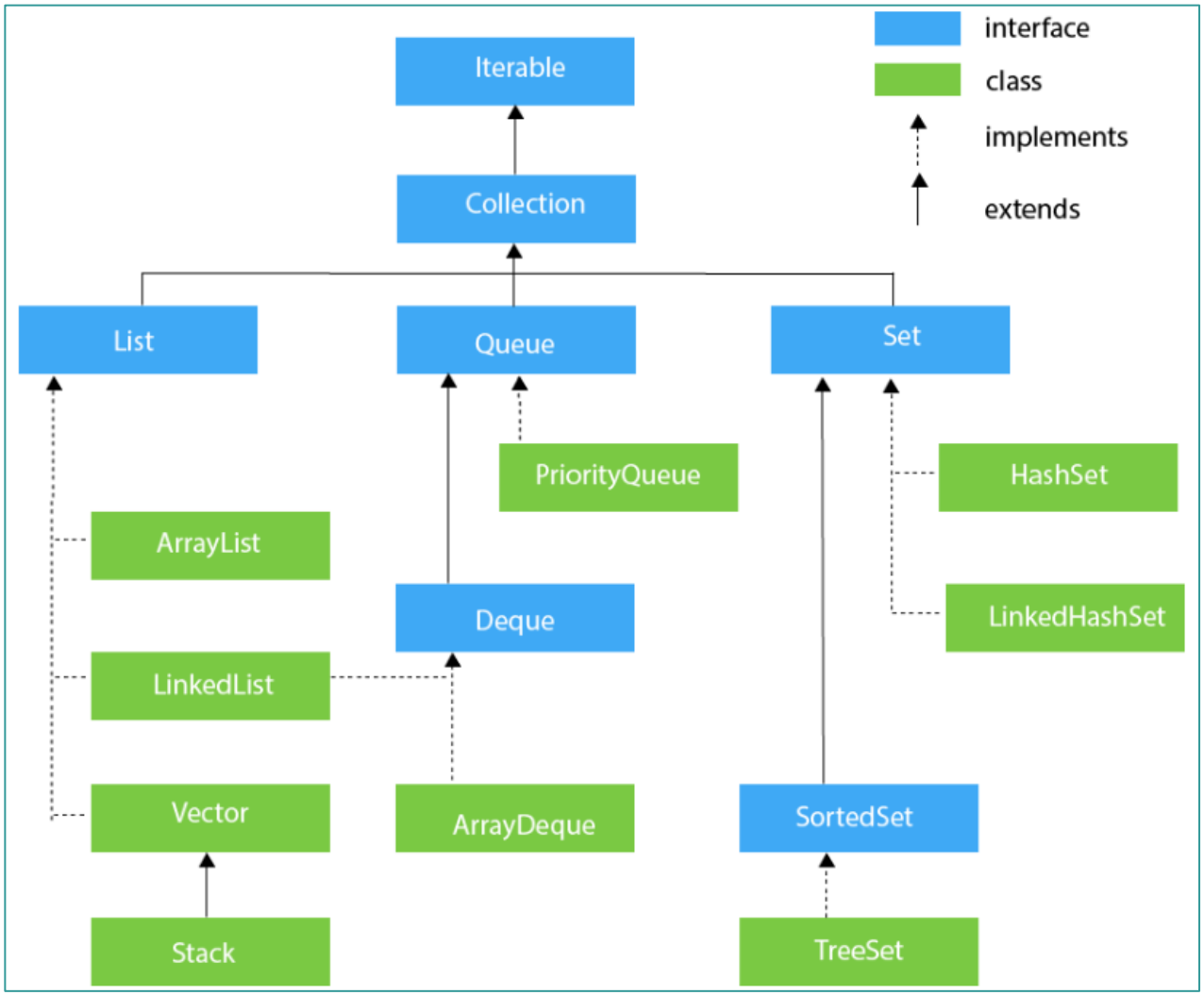
```
public interface Collection<E>
extends Iterable<E>
```

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any *direct* implementations of this interface: it provides implementations of more specific subinterfaces like `Set` and `List`. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

Bags or *multisets* (unordered collections that may contain duplicate elements) should implement this interface directly.

All general-purpose `Collection` implementation classes (which typically implement `Collection` indirectly through one of its subinterfaces) should provide two "standard" constructors: a void (no arguments) constructor, which creates an empty collection, and a constructor with a single argument of type `Collection`, which creates a new collection with the same elements as its argument. In effect, the latter constructor allows the user to copy any collection, producing an equivalent collection of the desired implementation type. There is no way to enforce this convention (as interfaces cannot contain constructors) but all of the general-purpose `Collection` implementations in the Java platform libraries comply.

Collections (2)



- Ιεραρχία κλάσεων Collections
- Το interface *Collection* κληρονομεί από το *Iterable* και επομένως όλες οι συλλογές υλοποιούν το interface Iterable και επομένως μπορούμε να τις διατρέξουμε με enhanced for που είναι μια μέθοδος του *Iterable* οι οποίες χρησιμοποιούν ένα *iterator* που είναι ένας δείκτης που διατρέχει τα αντικείμενα της συλλογής

Interface Collection

- Βασικές πράξεις που ορίζονται στο Collection
  - boolean add(E e)
  - boolean remove(Object element)
  - boolean contains(Object o)
  - int size()
  - boolean isEmpty()
  - <T> T[] toArray(T[] a)
  - Iterator iterator()
  - default Stream<E> stream()

Βασικά Collections

- Κάτω από το interface Collections η Java ορίζει τρία βασικά Collections
  - List
  - Queue
  - Set

Interface List (extends Collection)

- **Διατεταγμένη** συλλογή (ακολουθία) (ordered/indexed collection) που ξεκινά από το 0, όπως οι πίνακες
- Αυξάνουν και μειώνουν το μέγεθός τους δυναμικά καθώς προστίθενται ή διαγράφονται στοιχεία. Επιτρέπονται **διπλότυπα**
- Βασικές υλοποιήσεις
  - **ArrayList** (γίνεται backed από πίνακα)
  - **LinkedList** (γίνεται backed από δυναμική λίστα διπλής κατεύθυνσης)

ArrayList (implements List)

- Αυξάνουν και μειώνουν το μέγεθός τους δυναμικά καθώς προστίθενται ή διαγράφονται στοιχεία
- Διατεταγμένες ακολουθίες όπου το 1ο στοιχείο είναι το 0
- Επιτρέπουν διπλότυπα
- Δήλωση με generics

ArrayList πράξεις

- Βασικές πράξεις
  - boolean add(E e)
  - void add(int index, E element)
  - E get(int index)
  - E set(int index, E element)
  - E remove(int index)
  - boolean remove(Object o)
  - Boolean removeIf(Predicate)

interface Queue

- Το interface Queue είναι χρήσιμο για να υλοποιούμε ουρές (FIFO), στοίβες (LIFO) και ουρές προτεραιότητας (κλάση PriorityQueue)
- Το interface Deque (διαβάζεται Deck) κάνει extends το Queue και υλοποιείται από τις **LinkedList** (δυναμικά) και **ArrayDeque** (στατικά – resizable array όπως στο ArrayList)

Πράξεις τις Deque

FIFO (Ουρά / Queue) Methods	LIFO (Στοίβα / Stack) Methods
add(E e), addLast(E e) – προσθέτει στο τέλος της ουράς	push(E e) – προσθέτει στο τέλος της Στοίβας
E removeFirst(), E poll() – Αφαιρεί και επιστρέφει το 1ο στοιχείο της ουράς	E pop() – Αφαιρεί και επιστρέφει το τελευταίο στοιχείο της στοίβας)

- Η λογική First-In-First-Out (FIFO) αναφέρεται σε ουρές, ενώ η LIFO (Last0In-First-Out) σε στοίβες
- Και τα δύο μπορούν να υλοποιηθούν είτε με **LinkedList** ή με **ArrayDeque**

Queue - LinkedList (implements List, Deque)

- Διπλά συνδεδεμένη λίστα
- Υλοποιεί επιπλέον πράξεις που ορίζονται στο interface **Deque** (Double-ended queue)
  - E removeFirst(), poll(), pollFirst() – επιστρέφουν και διαγράφουν το 1ο στοιχείο της λίστας
  - E removeLast(), pollLast() – επιστρέφει και διαγράφει το τελευταίο στοιχείο της λίστας
  - void push() / E pop() -- Υλοποίηση στοίβας (με προσθήκη και αφαίρεση στην/από την αρχή της λίστας

Υλοποίηση Stack με ArrayDeck


```java
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.function.Consumer;

public class MyStack<T> {
    private final Deque<T> myStack = new ArrayDeque<>();

    public MyStack() {}

    public Deque<T> getMyStack() {
        return new ArrayDeque<>(myStack);
    }

   public void push(T t) {
        myStack.push(t);
   }

   public T pop() {
        return myStack.pop();
   }

   public void forEach(Consumer<T> action) {
        myStack.forEach(action);
   }
}
```

- Το ArrayDeck υλοποιεί το *Deque* και το *Queue*
- Υλοποιούμε ένα δικό μας My*Stack<T>* με δικό μας API για να μπορεί ο client να δημιουργεί λογική LIFO

Main


```java
package gr.aueb.cf.testbed.ch19;

public class MyStackApp {

    public static void main(String[] args) {
        MyStack<Integer> stack = new MyStack<>();

        stack.push(1);
        stack.push(2);
        int lastItem = stack.pop();

        System.out.println("Last inserted element: " + lastItem);

        stack.forEach(System.out::println);
    }
}
```

Υλοποίηση Queue με LinkedList


```
6   import java.util.function.Consumer;
7
8   public class MyQueue<T> {
9       private final Deque<T> myQueue = new LinkedList<>();
10
11      public MyQueue() {}
12
13      public Deque<T> getMyQueue() {
14          return new LinkedList<>(myQueue);
15      }
16
17      public void enQueue(T t) {
18          myQueue.addLast(t);
19      }
20
21      public T deQueue() {
22          return myQueue.poll();
23      }
24
25      public void forEach(Consumer<T> action) {
26          myQueue.forEach(action);
27      }
28  }
```

- Υλοποιούμε ένα δικό μας *Queue<T>* με δικό μας API για να μπορεί ο client να δημιουργεί λογική FIFO

Main


```
1   package gr.aueb.cf.testbed.ch19;
2
3   public class MyQueueApp {
4
5       public static void main(String[] args) {
6           MyQueue<String> myQueue = new MyQueue<>();
7
8           myQueue.enQueue("Red Car");
9           myQueue.enQueue("Green Car");
10          myQueue.enQueue("Blue Car");
11
12          String firstCar = myQueue.deQueue();
13          System.out.println("First in queue: " + firstCar);
14
15          myQueue.forEach(System.out::println);
16      }
17  }
```

class PriorityQueue

- Πρόκειται για Queue με ordering (natural ordering ή του Comparator στον υπερφορτωμένο constructor)
- Δεν περιέχει nulls
- Υλοποιείται με array
- Πράξεις
  - add(E e) για εισαγωγή στο τέλος
  - remove() / poll() για εξαγωγή του head

Διάσχιση Collection

- Διάσχιση (traverse) ενός Collection είναι η σειρά που εκτυπώνονται τα στοιχεία του
- Για παράδειγμα η διάσχιση στην ArrayList γίνεται κατά τη σειρά εισαγωγής, δηλ. από το στοιχείο 0 σειριακά μέχρι το size() - 1

Τρόποι διάσχισης

- Υπάρχουν οι τρεις τρόποι διάσχισης ενός collection στους οποίους αναφερθήκαμε και μαζί με την κλασσική for υπάρχουν τέσσερις συνολικά τρόποι διάσχισης μίας Collection, τρεις παλιοί και ένας καινούργιος
  - Με κλασσική *while ή for* από 0 – size()-1
  - Με *Iterator*
  - Με *enhanced for* (Java 5 και μετά)
  - Με aggregate methods της κλάσης Stream και τη μέθοδο *forEach()* από την Java 8 και μετά, που είναι και ο προτιμότερος τρόπος

Διάσχιση με forEach


```
myQueue.forEach(System.out::println);
```

- Η forEach είναι μία μέθοδος που δέχεται ως παράμετρο ένα Functional interface, το Consumer<T> που αντιπροσωπεύει μία πράξη που μπορούμε να κάνουμε με τα στοιχεία της λίστας (τα καταναλώνουμε) και δεν επιστρέφει τίποτα
- Το Consumer ορίζει μία μόνο μέθοδο, την void *accept (T t).* Επομένως το όρισμα μπορεί να υποκατασταθεί με *lambda* ή και με method reference

Interface Iterator

- Iterator<E>
- Χρησιμοποιείται για τη διάσχιση ενός Collection
- Η enhanced for και η forEach χρησιμοποιούν iterator
- Μας δίνει τη δυνατότητα να κάνουμε remove elements από ένα Collection

Διάσχιση με Iterator (1)

- Τον *Iterator* τον σκεφτόμαστε ως ένα **δείκτη** που δείχνει σε στοιχεία του Collection
  - Αρχικά, δείχνει πριν από το 1ο στοιχείο του Collection
  - Με την **hasNext()** δεν προχωράει αλλά απλά βλέπει αν υπάρχει επόμενο στοιχείο
  - Με την **next()** προχωράει
  - Με την **remove()** διαγράφει
- Ο *ListIterator* έχει επιπλέον πράξεις
  - hasPrevious()
  - previous()
  - add(E e)
  - set(E e)

Iterators

- Για να ξεκινήσουμε πρέπει να πάρουμε ένα Iterator από ένα Collection

```
10          List<String> cities = List.of("Athens", "Paris", "London");
11          Iterator<String> it = cities.iterator();
```

- Στη συνέχεια μπορούμε με μία while να διασχίσουμε το Collection

```
13          while (it.hasNext()) {
14              String s = it.next();
15              System.out.println(s);
16          }
17      }
```

Iterator και traverse με while


```java
package gr.aueb.cf.iterator;

import java.util.Iterator;
import java.util.List;

public class Main {

    public static void main(String[] args) {

        List<String> cities = List.of("Athens", "Paris", "London");
        Iterator<String> it = cities.iterator();

        while (it.hasNext()) {
            String s = it.next();
            System.out.println(s);
        }
    }
}
```

Enhanced for


```java
package gr.aueb.cf.iterator;

import java.util.List;

public class Main {

    public static void main(String[] args) {

        List<String> cities = List.of("Athens", "Paris", "London");

        for (String s : cities) {
            System.out.println(s);
        }
    }
}
```

- Η enhanced for χρησιμοποιεί iterator, under the hood

forEach()


```java
package gr.aueb.cf.iterator;

import java.util.List;

public class Main {

    public static void main(String[] args) {

        List<String> cities = List.of("Athens", "Paris", "London");

        cities.forEach(System.out::println);
    }
}
```

- Και η forEach χρησιμοποιεί iterator, under the hood

Iterator

- Με τη χρήση Iterator μπορούμε ταυτόχρονα με τη διάσχιση να διαγράψουμε στοιχεία της συλλογής με ασφάλεια με it.remove()
- Αν χρησιμοποιήσουμε μέσα σε διάσχιση την Collection.remove() τότε συμβαίνει σφάλμα (ConcurrentModificationException)

Διαγραφή με Iterator


```java
package gr.aueb.cf.iterator;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class Main {

    public static void main(String[] args) {

        List<String> cities = new ArrayList<>(List.of("Athens", "Paris", "London"));
        Iterator<String> it = cities.iterator();

        while (it.hasNext()) {
            if (it.next().equals("London")) {
                it.remove();
            }
        }

        cities.forEach(System.out::println);
    }
}
```

- Ασφαλής τρόπος διαγραφής με it.remove()

Διαγραφή με Collection.remove()


```
14          while (it.hasNext()) {
15              String s = it.next();
16              if (s.equals("London")) {
17                  cities.remove(s);
18              }
19          }
```

- Εδώ θα δημιουργηθεί exception γιατί μέσα σε iteration διαγράφουμε με Collection.remove() αντί για Iterator.remove()

```
Main
"C:\Program Files\Amazon Corretto\jdk11.0.10_9\bin\java.exe" "-javaagent:C:\Program
Exception in thread "main" java.util.ConcurrentModificationException Create breakpoint
    at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1043)
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:997)
    at gr.aueb.cf.iterator.Main.main(Main.java:15)

Process finished with exit code 1
```

Error κατά τη διαγραφή

- Βλέπουμε ότι ο Iterator *fails-fast* δηλαδή αμέσως μόλις πάει να γίνει διαγραφή ώστε να μην υπάρξει δομική (structural) ασυνέπεια στις εγγραφές της λίστας

Διαγραφή με Collection.removeIf


```java
package gr.aueb.cf.iterator;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class Main {

    public static void main(String[] args) {

        List<String> cities = new ArrayList<>(List.of("Athens", "Paris", "London"));

        cities.removeIf(s -> s.equals("London"));

        cities.forEach(System.out::println);
    }
}
```

- Η removeIf() είναι μία default μέθοδος του interface Collection. Μπορεί να χρησιμοποιηθεί με ασφάλεια αντί της iterator.remove()

Sets

- Συλλογές αντικειμένων
  - Του ίδιου τύπου
  - Δεν επιτρέπονται *διπλότυπα*
  - Δεν υπάρχει *διάταξη* (ordering) γιατί δεν υπάρχει indexing όπως στις λίστες
  - Το interface Set περιλαμβάνει μόνο μεθόδους που κληρονομούνται από το Collection interface

Set - HashSet

- Όπως η ArrayList είναι η πιο κοινή υλοποίηση του interface List, έτσι και το **HashSet** είναι η πιο κοινή υλοποίηση του Set
- Για παράδειγμα, αν εισάγουμε τους ακεραίους 2, 5, 7, 7, 12, 30, 30, το Set θα έχει τη μορφή {2, 5, 7, 12, 30}

HashSet vs List

- Τα HashSets έχουν καλύτερο χρόνο αναζήτησης από τις Λίστες, ιδιαίτερα καθώς μεγαλώνει το πλήθος των αντικειμένων που προστίθενται
- Τα Sets βασίζονται στην equals αφού δεν επιτρέπουν διπλότυπα, οπότε θα πρέπει να υπερφορτώνουμε την equals() (και την hashCode())
- Το HashSet βασίζεται στην hashCode() οπότε θα πρέπει να υπερφορτωθεί η hashCode() (και η equals())
- Δεν υπάρχει εγγύηση για τη σειρά διάσχισης

Διάσχιση HashSet


```java
package gr.aueb.cf.sets;

import java.util.HashSet;
import java.util.Set;

public class BagOfCoins {
    public static void main(String[] args) {
        Set<String> bag = new HashSet<>();
        bag.add("E1");
        bag.add("E2");
        bag.add("50cents");
        bag.add("1cent");

        bag.remove("1cent");

        bag.forEach(System.out::println);
    }
}
```

- Πρόκειται για set που γίνεται backed από HashMap
- Έχει σταθερούς O(1) χρόνους για add, remove, contains
- Επιτρέπει null

TreeSet

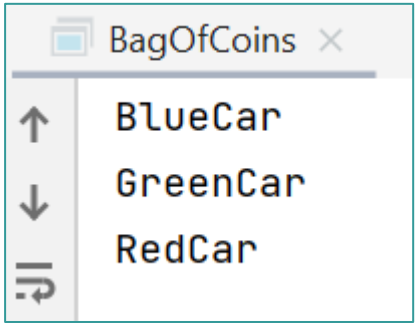
- Πρόκειται για Set αλλά με
  - Διάταξη (ordering) με βάση μία φυσική διάταξη (natural ordering) αν πρόκειται για built-in τύπους ή διάταξη που ορίζουμε εμείς υλοποιώντας το interface Comparable ή Comparator (σε κάθε περίπτωση θα πρέπει να υπάρχει ή να γίνεται override με συνέπεια η equals)
  - Σταθερός χρόνος log(n) για όλες τις βασικές πράξεις: add, remove, contains αφού πρόκειται για δένδρο αναζήτησης

Natural Ordering

- Φυσική κατάταξη με βάση την μαθηματική διάταξη αν πρόκειται για αριθμούς ή Ascii-based αν πρόκειται για χαρακτήρες ή Strings
- Αν πρόκειται για user-defined Κλάσεις πρέπει να ορίσουμε εμείς διάταξη υλοποιώντας το **interface Comparable, το οποίο είναι functional και ορίζει μόνο την μέθοδο int compareTo()**

TreeSet

```
1  package gr.aueb.cf.sets;
2
3  import java.util.*;
4
5  public class BagOfCoins {
6      public static void main(String[] args) {
7          Set<String> bag = new TreeSet<>();
8          bag.add("RedCar");
9          bag.add("GreenCar");
10         bag.add("BlueCar");
11         bag.add("1cent");
12
13         bag.forEach(System.out::println);
14     }
15 }
```



- Φυσική Κατάταξη για *Strings*
- Η κλάση **String** υλοποιεί την *compareTo()* με βάση την οποία γίνεται η σύγκριση και κατάταξη των στοιχείων

Ταξινόμηση αναλυτικά

- Στη συνέχεια θα εξετάσουμε τα interfaces **Comparable** και **Comparator** που χρησιμοποιούνται **στην ταξινόμηση αντικειμένων τύπου List, όπως η ArrayList**
- Θα δούμε πιο αναλυτικά παρακάτω πως μπορούμε να ταξινομούμε μία ArrayList (και γενικά μία List)

Ταξινόμηση

- Η ταξινόμηση μπορεί να γίνει με τη μέθοδο **Collections.sort()** της κλάσης **Collections**. Σύμφωνα με το Javadoc η μέθοδος sort() της κλάσης **Collections** ταξινομεί μία λίστα, είτε: (1) σε αύξουσα σειρά σύμφωνα με το **Comparable** που θα πρέπει να έχουμε ορίσει, είτε (2) σύμφωνα με τον **Comparator** που θα πρέπει να έχουμε ορίσει
  - ***Collections.sort(List<T> list)***, όπου *list* μία *List* Ταξινομεί με βάση το natural ordering του T (Comparable)
  - ***Collections.sort(List<T> list, Comparator<? super T> c)***, όπου *list* μία *List* και *c* ένας Comparator. Ταξινομεί με βάση τον Comparator
- Η ταξινόμηση μπορεί επίσης να γίνει με τη μέθοδο List.sort(Comparator<? Super E> c) του interface List
  - ***List.sort(Comparator<? super E> c)***

Comparable

- Το **Interface Comparable** είναι functional Interface και ορίζει μία μόνο μέθοδο την **compareTo()**
- Κάνοντας implement το **Interface Comparable** σε μία κλάση δηλ. υλοποιώντας την **compareTo(A a)** μπορούμε να ορίσουμε ***μόνο μία ταξινόμηση, που αναφέρεται ως φυσική κατάταξη (natural ordering)*** των αντικειμένων της κλάσης, δηλ. αντίστοιχη με την κατάταξη που παρέχεται από το JVM: οι αριθμοί κατατάσσονται αριθμητικά –δηλ. το 1 πριν το 10 κλπ, οι χαρακτήρες και τα **String** λεξικογραφικά (ASCII-betically) δηλ. το a είναι πριν g κλπ.

Comparable με String


```java
package gr.aueb.cf.sorting;

public class Product implements Comparable<Product> {
    private String description;
    private double price;
    private int quantity;

    public Product() {}

    public Product(String description, double price, int quantity) {...}

    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description; }
    public double getPrice() { return price; }
    public void setPrice(double price) { this.price = price; }
    public int getQuantity() { return quantity; }
    public void setQuantity(int quantity) { this.quantity = quantity; }
    @Override
    public String toString() {...}

    @Override
    public int compareTo(Product o) {
        return this.description.compareTo(o.description);
    }
}
```

- Όταν συγκρίνουμε **πεδία String** όπως στο παράδειγμα που ταξινομούμε με βάση το **description** μπορούμε απλώς να επιστρέφουμε το αποτέλεσμα που επιστρέφει η μέθοδος **compareTo** για Strings
- Αυτός ο τρόπος κατάταξης που είναι ***αύξουσα κατάταξη (ascending ordering)*** όπως είπαμε ονομάζεται ***φυσική κατάταξη (natural ordering)*** γιατί συμβαδίζει με την λογική των μαθηματικών, και του JVM.

Comparable με int


```java
package gr.aueb.cf.sorting;

public class Product implements Comparable<Product> {
    private String description;
    private double price;
    private int quantity;

    public Product() {}

    public Product(String description, double price, int quantity) {...}

    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description; }
    public double getPrice() { return price; }
    public void setPrice(double price) { this.price = price; }
    public int getQuantity() { return quantity; }
    public void setQuantity(int quantity) { this.quantity = quantity; }
    @Override
    public String toString() {...}

    @Override
    public int compareTo(Product o) {
        return this.quantity - o.quantity;
    }
}
```

- Όταν συγκρίνουμε **πεδία int** όπως στο παράδειγμα που ταξινομούμε με βάση το **quantity** μπορούμε απλώς να επιστρέφουμε το αποτέλεσμα της αφαίρεσης, που συμβαδίζει με τη λογική της compareTo

Comparable με double


```java
package gr.aueb.cf.sorting;

public class Product implements Comparable<Product> {
    private String description;
    private double price;
    private int quantity;

    public Product() {}

    public Product(String description, double price, int quantity) {...}

    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description; }
    public double getPrice() { return price; }
    public void setPrice(double price) { this.price = price; }
    public int getQuantity() { return quantity; }
    public void setQuantity(int quantity) { this.quantity = quantity; }
    @Override
    public String toString() {...}

    @Override
    public int compareTo(Product o) {
        return Double.compare(this.price, o.price);
    }
}
```

- Όταν συγκρίνουμε **πεδία double** όπως στο παράδειγμα που ταξινομούμε με βάση το **price** μπορούμε να επιστρέφουμε το αποτέλεσμα της Double.compare

Main


```java
package gr.aueb.cf.sorting;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Main {

    public static void main(String[] args) {
        List<Product> products = new ArrayList<>(List.of(new Product("Eggs", 3.5, 5),
                new Product("Apples", 4.5, 9), new Product("Oranges", 2.5, 2),
                new Product("Milk", 8.5, 4)
        ));

        Collections.sort(products);
        products.forEach(System.out::println);

        System.out.println();

        Collections.reverse(products);
        products.forEach(System.out::println);

    }
}
```

- Με Collections.sort ταξινομούμε με βάση το Comparable
- Με Collections.reverse ταξινομούμε με βάση αντίστροφη κατάταξη

Comparator

- Με το Comparable μπορούμε να ταξινομήσουμε **μόνο ως προς ένα πεδίο**
- Για επιπλέον ταξινομήσεις και με άλλα πεδία θα πρέπει να χρησιμοποιούμε (και να κάνουμε implement) το **interface Comparator**
- Το **interface Comparator** είναι Functional Interface και με μία μόνο abstract μέθοδο, την **compare(A a1, A a2)**, όπου **a1**, **a2** είναι αντικείμενα του τύπου A που θέλουμε να ταξινομήσουμε
- Παρατηρήστε ότι η **compare(A a1, A a2)** λαμβάνει ως είσοδο δύο αντικείμενα **a1**, **a2** και μπορεί να υλοποιείται από μία ανεξάρτητη κλάση που κάνει implements τον **Comparator**

Comparator natural Order / reverseOrder


```java
package gr.aueb.cf.sorting;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class Main {

    public static void main(String[] args) {
        List<Product> products = new ArrayList<>(List.of(new Product("Eggs", 3.5, 5),
                new Product("Apples", 4.5, 9), new Product("Oranges", 2.5, 2),
                new Product("Milk", 8.5, 4)
        ));

        Collections.sort(products, Comparator.naturalOrder());
        products.forEach(System.out::println);

        System.out.println();

        Collections.sort(products, Comparator.reverseOrder());
        products.forEach(System.out::println);
    }
}
```

- Το ίδιο που κάναμε με το Comparable μπορούμε να το κάνουμε με το Comparator με τις static μεθόδους *naturalOrder()* και *reverseOrder*() όπως φαίνεται στο παράδειγμα

List.sort()


```java
package gr.aueb.cf.sorting;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class Main {

    public static void main(String[] args) {
        List<Product> products = new ArrayList<>(List.of(new Product("Eggs", 3.5, 5),
                new Product("Apples", 4.5, 9), new Product("Oranges", 2.5, 2),
                new Product("Milk", 8.5, 4)
        ));

        products.sort(Comparator.naturalOrder());
        products.forEach(System.out::println);

        System.out.println();

        products.sort(Comparator.reverseOrder());
        products.forEach(System.out::println);

        products.sort(Comparator.naturalOrder());
    }
}
```

- Το ίδιο όπως πριν αλλά με List.sort

Comparator.comparing (1)

- Η comparing(sortKeyFunction) είναι μία static μέθοδος του Comparator interface που λαμβάνει ως είσοδο ένα Function interface που επιστρέφει ένα πεδίο ή γενικά key με το και ταξινομεί με βάση αυτό το sort key
- Για παράδειγμα, αν θέλουμε να ταξινομήσουμε με βάση το description μπορούμε να το κάνουμε όπως παρακάτω, περνώντας το *Product::getDescription,* που είναι method reference του getter του πεδίου description, ως παράμετρο της comparing

```
products.sort(Comparator.comparing(Product::getDescription));
products.forEach(System.out::println);
```

Comparator.comparing (2)


```
products.sort(Comparator.comparing(Product::getPrice).thenComparing(Product::getQuantity));
products.forEach(System.out::println);
```

```
products.sort(Comparator.comparing(Product::getPrice).thenComparing(Product::getQuantity).reversed());
products.forEach(System.out::println);
```

- Στο 1ο παράδειγμα ταξινομούμε ως προς την price και μετά, αν υπάρχουν ίδια prices, ως προς το quantity
- Στο 2ο παράδειγμα είναι το ίδιο αλλά τα αποτελέσματα είναι σε φθίνουσα ταξινόμηση ως προς το getPrice

Comparator.comparing (3)


```
products.sort(Comparator.comparing(Product::getDescription)
        .thenComparingDouble(Product::getPrice)
        .thenComparingInt(Product::getQuantity));
products.forEach(System.out::println);
```

- Επίσης παρέχονται και πιο ειδικές comparing, όπως comparingInt και comparingDouble

Comparator.comparing (4)


```
products.sort(Comparator.comparing(Product::getDescription)
        .thenComparing(Product::getPrice, (a1, a2) -> Double.compare(a2, a1))
        .thenComparingInt(Product::getQuantity));

products.forEach(System.out::println);
```

- Αν θέλουμε φθίνουσα ταξινόμηση για κάποιο συγκεκριμένο πεδίο και όχι το αρχικό, τότε μπορούμε να δώσουμε $2^{η}$ παράμετρο στην comparing που να είναι ένας Comparator που επενεργεί επί του sort-key δηλαδή επί του price. Εδώ ο Comparator υλοποιείται με ένα lambda που υλοποιεί την compareTo
- Εδώ για παράδειγμα θέλουμε για όμοια descriptions η τιμή να εμφανίζεται σε φθίνουσα κατάταξη και για αυτό δίνουμε Double.compare(a2, a1) και όχι το default που είναι compareTo(a1, a2). Τα a1, a2 είναι δύο συνεχόμενα στοιχεία της λίστας

Comparator.comparing (5)


```
1   package testbed.tmp;
2
3   import java.util.ArrayList;
4   import java.util.Comparator;
5   import java.util.List;
6
7   public class Main {
8
9       public static void main(String[] args) {
10          List<Product> products = new ArrayList<>(
11                  List.of(new Product("Milk", 10.5, 10),
12                          new Product("Honey", 20.5, 20),
13                          new Product("Honey", 30.5, 30),
14                          new Product("Milk", 90.5, 8)));
15
16          products.sort(Comparator.comparing(Product::getDescription)
17                  .thenComparing(Product::getPrice, Comparator.reverseOrder())
18                  .thenComparing(Product::getQuantity));
19
20          products.forEach(System.out::println);
21      }
22  }
```

- Εναλλακτικά μπορούμε ως $2^{η}$ παράμετρο στην comparing να περάσουμε Comparator που επενεργεί επί του sort key (βλ. γραμμή17)

Streams API

- Το Streams API μας δίνει τη δυνατότητα functional programming στα Collections. Φεύγουμε από τη λογική της for και πάμε στη λογική του functional programming με callbacks
- Αυτή τη δυνατότητα μας τη δίνει το interface Stream<T> του package java.util.stream
- Τα streams παίρνουν input από τα Collections ή Arrays και επενεργούν επί αυτών χωρίς να αλλάζουν (modify) το αρχικό data structure
- Επιστρέφουν ένα καινούργιο collection ή μία τιμή

Intermediate operations

- Τα Streams παρέχουν ένα API με **intermediate operations** που μπορούμε να κάνουμε invoke με chaining, όπως **filter** (που φιλτράρει το collection με βάση ένα Predicate), **map** (που αντιστοιχεί τα στοιχεία του Collection σε μια άλλη τιμή με ένα callback), **sorted** (που ταξινομεί με βάση ένα Comparator)

Terminal operations

- Επίσης, παρέχονται μέθοδοι που καταναλώνουν το output των ενδιάμεσων μεθόδων όπως **collect** (που εξάγει τα αποτελέσματα σε List, Set κλπ.), **forEach** (που πάλι καταναλώνει τα δεδομένα εξόδου και τα επεξεργάζεται ένα-ένα), **reduce** (που καταναλώνει τα δεδομένα εξόδου και εξάγει μία τιμή, .π.χ. άθροισμα), **findFirst**() που επιστρέφει ένα Optional<T> με το πρώτο στοιχείο (αν υπάρχει αλλιώς empty Optional)

Filter / collect


```
public static void main(String[] args) {
    List<Product> products = new ArrayList<>(List.of(new Product("Eggs", 3.5, 5),
            new Product("Apples", 4.5, 9), new Product("Apples", 5.5, 6), new Product("Oranges", 4.5, 2),
            new Product("Milk", 8.5, 4)
    ));

    List<Product> eggs = products.stream()
                    .filter(product -> product.getDescription().equals("Eggs"))
                    .collect(Collectors.toList());

    eggs.forEach(System.out::println);
```

- Η **filter** επιστρέφει τον ίδιο τύπο stream με βάση ένα Predicate (boolean function)
- Η collect επιστρέφει ένα collection με βάση τον Collector

Sorted / forEach


```
16          List<Product> apples = products.stream()
17                          .filter(product -> product.getDescription().equals("Apples"))
18                          .collect(Collectors.toList());
19
20          apples.stream()
21                  .sorted(Comparator.comparingInt(Product::getQuantity))
22                  .forEach(System.out::println);
```

- Η sorted λειτουργεί όπως η Collections.sort και επιστρέφει Stream ενώ η forEach λειτουργεί όπως η κλασική forEach

Collect


```java
package testbed.tmp;

import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;
import java.util.stream.Collectors;

public class SetTest {

    public static void main(String[] args) {
        Set<String> cities = new HashSet<>();
        cities.add("Athens");
        cities.add("London");
        cities.add("Boston");

        Set<String> nonGreekCities = cities.stream()
                .filter((c) -> !c.equals("Athens"))
                .sorted()
                .collect(Collectors.toCollection(TreeSet::new));

        nonGreekCities.forEach(System.out::println);
    }
}
```

- Το Set δεν μπορεί να γίνει sorted και για αυτό εξάγουμε σε TreeSet (ή LinkedHashSet)
- Η collect δημιουργεί ένα νέο instance του result container, μετά εισάγει τα στοιχεία και τέλος τα κάνει merge στο νέο collection

Map (1)


```
10      public static void main(String[] args) {
11          List<Product> products = new ArrayList<>(
12                  List.of(new Product("Milk", 10.5, 10),
13                          new Product("Honey", 20.5, 20),
14                          new Product("Honey", 30.5, 30),
15                          new Product("Eggs", 90.5, 8)));
16
17          List<Double> prices = products.stream() Stream<Product>
18                  .map(Product::getPrice) Stream<Double>
19                  .collect(Collectors.toList());
20
21          prices.forEach(System.out::println);
```

- Η map αντιστοιχεί τα δεδομένα εισόδου σε τιμή / τιμές εξόδου σύμφωνα με μία παράμετρο τύπου ***Function*** που είναι Functional Interface

Map (2)


```
10  public static void main(String[] args) {
11      List<Product> products = new ArrayList<>(
12              List.of(new Product("Milk", 10.5, 10),
13                      new Product("Honey", 20.5, 20),
14                      new Product("Honey", 30.5, 30),
15                      new Product("Eggs", 90.5, 8),
16                      new Product("Oranges", 90.5, 8),
17                      new Product("Oranges", 90.5, 16)));
18
19      List<Product> updatedPriceProducts = products.stream()
20              .map(p -> new Product(p.getDescription(), p.getPrice() + p.getPrice() * 0.12, p.getQuantity()))
21              .collect(Collectors.toList());
22      updatedPriceProducts.forEach(System.out::println);
```

- Η map μπορεί να επιστρέφει και νέα αντικείμενα, όπως για παράδειγμα new Product με τιμές με βάση τις τιμές εισόδου του Stream

Reduce / mapToInt / mapToDouble


```
16      int orangesTotalCount = products.stream()
17              .filter(p -> p.getDescription().equals("Oranges"))
18              .reduce(0, (total, e) -> total + e.getQuantity(), Integer::sum);
19
20      int orangesTotalCount2 = products.stream()
21              .filter(p -> p.getDescription().equals("Oranges"))
22              .mapToInt(Product::getQuantity)
23              .sum();
24
25      double orangesPriceTotal = products.stream()
26              .filter(p -> p.getDescription().equals("Oranges"))
27              .mapToDouble(Product::getPrice)
28              .sum();
```

- Η reduce επιστρέφει μία τιμή επενεργώντας επί των δεδομένων εισόδου. Η 1η παράμετρος είναι η αρχική τιμή και οι παράμετροι υπονοείται ότι είναι ίδιου τύπου με τις παραμέτρους εισόδου (στοιχεία του stream). Αν δεν είναι, τότε χρειαζόμαστε και μία 3η παράμετρο
- Μπορούμε να απλοποιήσουμε με την mapToInt (ή mapToDouble αντίστοιχα)

Streams API

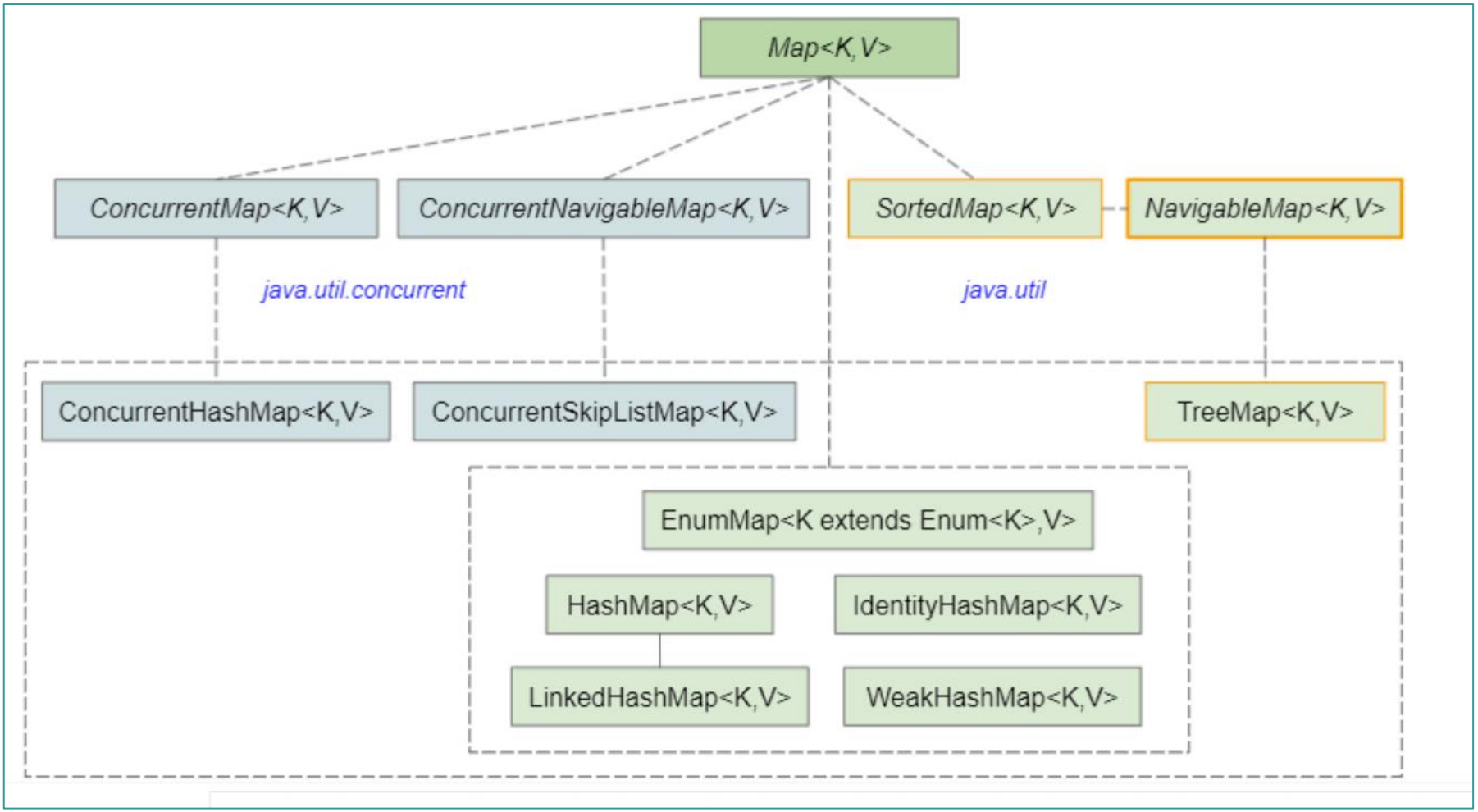

```java
package gr.aueb.elearn.ch7.streams;

import gr.aueb.elearn.ch7.compare.Product;

import java.util.*;
import java.util.stream.Collectors;

public class ProductsStream {

    public static void main(String[] args) {
        List<Product> products = products = Arrays.asList(
                new Product("Milk", 2.20, 2),
                new Product("Eggs", 1.5, 5),
                new Product("Honey", 8.30, 2));

        List<String> sortedProductsByPrice = products.stream()
                .filter(product -> product.getQuantity() == 2)
                .sorted(Comparator.comparing(Product::getDescription))
                .map(Product::getDescription)
                .collect(Collectors.toList());

        sortedProductsByPrice.forEach(System.out::println);
    }
}
```

- Όπως πριν αλλά με περισσότερες επιλογές στην stream()
- Η *filter* λαμβάνει ως είσοδο ένα αντικείμενο τύπου Predicate –δηλ. μία συνθήκη boolean που ορίζουμε με lambda-

Maps

- Οι χάρτες (maps) είναι μια συλλογή από *map entries*
- Κάθε map entry συνδέει ένα πεδίο κλειδί (Key) με μια τιμή (Value)
- Κάθε Key *πρέπει να είναι μοναδικό* (ή μόνο ένα null) *και να οδηγεί σε μοναδική τιμή*.
- Η βασική υλοποίηση του interface Map είναι το **HashMap**

Ιεραρχία Map



Maps – Βασικές Πράξεις

- Όπως τα Sets, έτσι και **τα Maps δεν έχουν διάταξη**
- Βασικές Πράξεις
  - *V **put(K key, V value)** – εισάγει ένα map-entry δηλαδή ένα ζεύγος κλειδιού-τιμής*
  - *V **get(Object key)** – επιστρέφει την τιμή που αντιστοιχεί στο key ή null*

  Όπου K είναι ο τύπος του κλειδιού και V ο τύπος των mapped values
- Ως values μπορούν να είναι nulls

Περισσότερες μέθοδοι

- *boolean* ***containsKey(Object Key)***
- *boolean* ***containsValue(Object value)***
- *Set<Map.Entry<K,V>>* ***entrySet()*** *– επιστρέφει τα map entries*
- *Set<K>* ***keyset()*** -- επιστρέφει ένα Set view των keys του Map
- *Collection<V>* ***values()*** *-- επιστρέφει τα values*
- *V* ***remove(Object key)***
- *int* ***size()***

interface Map.Entry<K,V>

- ***Map.Entry*** – Πρόκειται για static inner interface του *interface Map* που περιλαμβάνει ένα ζεύγος key/value. Οι βασικές πράξεις που ορίζει είναι:
  - *V* ***getValue()***
  - *K* ***getKey()***
  - *V setValue(V value)*
  - *int hashCode() – επιστρέφει το hashCode του entry*
  - *boolean equals(Object o)*

Μέθοδος Map.entrySet()

- ***Map.entrySet()*** – όπως αναφέρθηκε, επιστρέφει ένα Set view των map-entries του Map
- Οι αλλαγές στο entry*Set* γίνονται reflect στο *Map* και το αντίστροφο

Διάσχιση Map

- Με iterator στο entrySet
- Μέσω το entrySet()
- Μέσω του keyset() ή του values()
- Με forEach() στο map

HashMap


```
1   package gr.aueb.elearn.ch8.countries;
2
3   import java.util.HashMap;
4   import java.util.Iterator;
5   import java.util.Map;
6
7   public class MapDemo {
8
9       public static void main(String[] args) {
10          Map<String, String> countries = new HashMap<>();
11
12          countries.put("GR", "Greece");
13          countries.put("FR", "France");
14          countries.put("USA", "United States of America");
15          countries.put("IT", "Italy");
```

- Η βασική υλοποίηση του interface Map, είναι το HashMap. Παραπάνω ορίζουμε ένα HashMap και εισάγουμε στοιχεία

Διάσχιση με Iterator


```
17      Iterator<Map.Entry<String, String>> it = countries.entrySet().iterator();
18      while (it.hasNext()) {
19          Map.Entry<String, String> entry = it.next();
20          System.out.println(entry.getKey() + "" + entry.getValue());
21      }
```

- Ο *Iterator it* δείχνει σε map-entries του *Set*
- Με *getKey()* και *getValue()* λαμβάνουμε το κλειδί και την τιμή κάθε map-entry

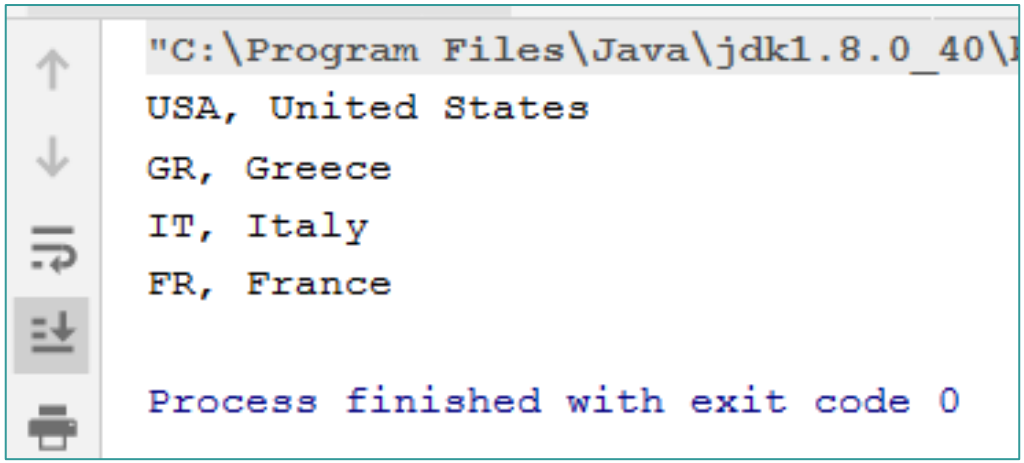
Διάσχιση με enhanced for


```
23      for (Map.Entry<String, String> entry : countries.entrySet()) {
24          System.out.println(entry.getKey() + "" + entry.getValue());
25      }
```

- Η enhanced for χρησιμοποιεί iterator εσωτερικά, ωστόσο δεν έχουμε άμεση πρόσβαση στον iterator

Διάσχιση με forEach


```
27        countries.forEach((k, v) -> System.out.println(k + ":" + v));
```



- Τα αποτελέσματα είναι τα παραπάνω
- Η διάσχιση δεν εγγυάται κάποιο συγκεκριμένο ordering

HashMap vs ArrayList

- Η ArrayList έχει χρόνο αναζήτησης γραμμικό, O(n)
- Αν έχουμε πολλές αναζητήσεις αποδοτικότερη δομή δεδομένων είναι το HashMap, που ιδανικά έχει χρόνο αναζήτησης O(1)

HashMap σε λεπτομέρεια

- Η δομή HashMap είναι μία δομή δεδομένων που κάθε κόμβος της περιέχει δύο στοιχεία: (i) το ***κλειδί*** αναζήτησης (***Key***) , και (ii) την ***τιμή*** (***Value***) με την οποία σχετίζεται το κλειδί
- Τόσο το κλειδί, όσο και το value πρέπει να είναι κλάσεις και όχι πρωταρχικοί τύποι

HashMap παραδείγματα

- Έστω μία δομή **`HashMap<K,V>`**,όπου **`K`** είναι η κλάση που αναπαριστά το κλειδί (ΚΕΥ) και **`V`** η κλάση που αναπαριστά το Value
- Παραδείγματα θα μπορούσαν να είναι <AFM, Forologoumenos> ή <AMKA, Asfalismenos> ή <Mhtrwo, Ekpaideyomenos> , κλπ

HashMap και nulls

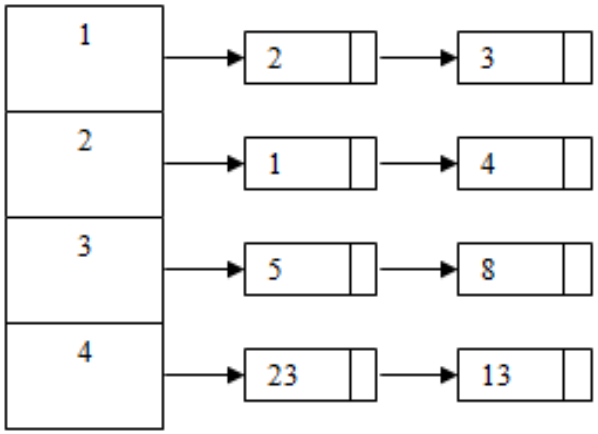
- Οι τιμές (values) σε ένα HashMap μπορεί να είναι και null
- Επίσης, μπορεί να υπάρχει μέσα στο HashMap **ένα μόνο null κλειδί** (μιας και δεν επιτρέπονται διπλότυπα δηλ. διπλά κλειδιά)

Κλειδιά και immutability

- Ο τύπος των κλειδιών ενός Map συνίσταται να είναι immutable (π.χ. String) ώστε να μην μπορεί να αλλάξει ένα κλειδί και υπάρχει ασυνέπεια στο Map
- Γιαυτό το λόγο συνηθισμένες κλάσεις για τύπο κλειδιού είναι String και Integer

Υλοποίηση HashMap

- Ένα HashMap υλοποιείται με μία δομή παρόμοια με λίστα γειτνίασης. Δηλαδή ένα HashMap υλοποιείται με ένα πίνακα **κάθε στοιχείο του οποίου αντιστοιχεί σε ένα bucket**. ***Κάθε διαφορετικό bucket έχει διαφορετικό hash code value***
- **Κάθε bucket δείχνει σε μία γραμμική λίστα,** ενώ από ένα σημείο και μετά (αν υπάρχουν περισσότερα από οκτώ entries) η λίστα μετατρέπεται σε δένδρο αναζήτησης, όπου κάθε κόμβος έχει διαφορετικό Key με ίδιο όμως hash value



HashMap Buckets

- ***Κάθε bucket του HashMap αντιστοιχεί σε ένα HashCode*** και κάθε κόμβος της λίστας έχει το ίδιο hashCode αλλά διαφορετικό κλειδί
- Αυτή η δυνατότητα, ***να μπορούν δηλαδή διαφορετικά κλειδιά να δίνουν το ίδιο hash value*** καλείται ***collision*** και οφείλεται στο ότι το εύρος των τιμών εισόδου είναι γενικώς απεριόριστο ενώ οι τιμές εξόδου μέσω της hash function στον πίνακα των bucket είναι περιορισμένες από τον αριθμό των θέσεων του πίνακα με αποτέλεσμα να δημιουργούνται collisions
- Δεν θα θέλαμε πολλά collisions γιατί χειροτερεύει ο χρόνος αναζήτησης. Χωρίς collisions ο χρόνος αναζήτησης είναι σταθερός O(1)

Πολυπλοκότητες χρόνου (1)

- Ιδανικά θα θέλαμε κάθε map-entry να αποθηκεύεται σε ένα bucket ώστε ο χρόνος αναζήτησης να είναι O(1) (δηλ. σταθερός χρόνος απλά για τον υπολογισμού του hash-value)
- Αν όμως έχουμε ένα πίνακα n θέσεων και k > n κλειδιά, τότε για να αποθηκευτούν τα entries δημιουργούνται collisions, δηλαδή διαφορετικά entries τοποθετούνται στο ίδιο bucket γιατί αν και έχουν διαφορετικό hashCode, το HashMap κάνει το τελικό mapping (οπότε αν έχουμε 16 θέσεις και 17 entries, το τελευταίο entry θα πάει σε υπάρχουσα θέση).

HashCodes (1)

- Η τοποθέτηση σε buckets των entries γίνεται με μια διαδικασία modulus του hash-value με το array size
- Αν για παράδειγμα το hashCode ενός key είναι 5 και το HashTable έχει μήκος 10 (από 0 - 9), τότε γίνεται 5 % 10 και το entry τοποθετείται στη θέση 5
- Αν το hashcode είναι 21, γίνεται 21 % 10 και το entry τοποθετείται στη θέση 1

HashCodes (2)

- Αν είχαμε και ένα άλλο entry με key = 1, και hashCode = 1, τότε 1 % 10 = 1 και θα είχαμε collision αφού και το 21 έχει το ίδιο modulus
- Σε αυτή την περίπτωση έχουμε στο ίδιο bucket, δύο entries και αυξάνεται ο χρόνο αναζήτησης

Πολυπλοκότητες χρόνου

- Ο τρόπος που τοποθετούνται διαφορετικά entries στο ίδιο bucket είναι μέσα σε μια λίστα η οποία δίνει γραμμικό χρόνο αναζήτησης.
- Από την Java 8 και μετά, μετά από οκτώ στοιχεία η λίστα αναδιοργανώνεται σε δένδρο και δίνει O(logn) χρόνο αναζήτησης

Initial Capacity και Load Factor

- Ο default constructor του HashMap δημιουργεί ένα πίνακα με initial capacity 16 buckets (το μέγεθος του πίνακα πρέπει να είναι πολλαπλάσιο του 2) και load factor 0.75
- Αυτό σημαίνει πως όταν εισαχθούν 12 entries (16*0.75 = 12) τότε γίνεται rehashing δηλαδή ο πίνακας διπλασιάζει το μέγεθός του ώστε να ελαχιστοποιείται η πιθανότητα collisions
- Καλό θα ήταν αν γνωρίζαμε πόσες εγγραφές θα εισαχθούν στον πίνακα κατά μέγιστο (έστω n) να ορίζαμε τόσο initial capacity ώστε το initialCapacity*loadFactor > n, έτσι ώστε να χωρούν όλες οι εγγραφές χωρίς rehashing
- Αν για παράδειγμα είχαμε n=1000, τότε το initialCapacity θα πρέπει να είναι μεγαλύτερο από 1000/0.75=1334
- Τότε θα μπορούσαμε να αρχικοποιήσουμε το HashMap με τον υπερφορτωμένο constructor σε **`HashMap<>(1334, 0.75F)`** ή σκέτο **`HashMap<>(1334)`** αφού το **`0.75F`** είναι το default

Χρήσιμες μέθοδοι της HashMap

- **`V put(K, V)`** – ***εισάγει*** στο HashMap τον κόμβο (K, V). Επιστρέφει null αν το K δεν υπήρχε (ή υπήρχε και ήταν null) ή το προηγούμενο value V, αν το K υπήρχε και είχε value, **οπότε σε αυτή την περίπτωση *αντικαθιστά (replace)* το value**.
  - Κατά την εισαγωγή πρώτα καλείται η `hashCode(K)` και αν με βάση το hash code το hashCode mod mapArraySize υπάρχει στο HashMap (υπάρχει δηλ. ήδη bucket με αυτή την τιμή) τότε δημιουργείται collision. Στη συνέχεια καλείται η `equals()` και αν στο συγκεκριμένο bucket υπάρχει το κλειδί που πρόκειται να εισαχθεί, το value αντικαθίσταται, αν όχι εισάγεται νέος κόμβος. ***Προϋπόθεση είναι να έχουν γίνει σωστά override και η `hashCode()` και η `equals()` της κλάσης K***.
  - Αν δεν έχουν γίνει override οι μέθοδοι `hashCode()` και `equals()` τότε η `hashCode()` επιστρέφει το memory location (τον δείκτη δηλαδή) του αντικειμένου `K`, και επομένως για κάθε key δημιουργείται νέο bucket γιατί η `hashCode()` επιστρέφει διαφορετικό memory location για κάθε νέο αντικείμενο `K`, παρότι μπορεί το `K` να είναι το ίδιο
  - Επομένως μπορεί **δύο διαφορετικά αντικείμενα αλλά με ίδια κλειδιά να πάνε σε διαφορετικά buckets** και μετά ***δεν δουλεύει σωστά το `HashMap`*** αφού η `equals()` δεν καλείται ποτέ (η `equals()` δουλεύει μόνο στο πλαίσιο του bucket)

HashMap.put(K, V)

- Αν έχει γίνει override η **`hashCode(K)`** αλλά όχι η **`equals()`**, τότε ναι μεν θα βρεθεί το σωστό bucket αν υπάρχει ίδιο ΚΕΥ, αλλά στη συνέχεια **η `equals()` δεν θα δώσει ισότητα** γιατί τα αντικείμενα θα είναι διαφορετικά (η μη-υπερφορτωμένη **`equals()`** είναι η **`==`**), οπότε ακόμα και στην περίπτωση που θα υπάρχει κόμβος με ίδιο ΚΕΥ δεν θα αντικατασταθεί το value του κόμβου αυτού αλλά θα δημιουργηθεί νέος κόμβος με το ίδιο κλειδί

Χρήσιμες μέθοδοι της HashMap (1)

- **`V putIfAbsent(K, V)`** – εισάγει στο HashMap τον κόμβο (K, V). Επιστρέφει null αν το K δεν υπήρχε ή υπήρχε και ήταν null ή το προηγούμενο value V, αν το K υπήρχε
- **`V get(K)`** – επιστρέφει το Value του K
  - Κατά την αναζήτηση υπολογίζεται το `hasCode(K)` και αν το hash value υπάρχει σε κάποιο bucket, τότε ελέγχεται με την `equals()` αν υπάρχει κόμβος με ίδιο κλειδί K. Αν υπάρχει, επιστρέφεται το Value, αλλιώς επιστρέφει null
- **`boolean containsKey(K)`** – true αν το κλειδί υπάρχει, αλλιώς false
- **`boolean containsValue(V)`** – true αν το value υπάρχει, αλλιώς false

Χρήσιμες μέθοδοι της HashMap (2)

- **`boolean replace(Key, oldValue, newValue)`** - αντικαθιστά το mapping αν υπάρχει το **`oldValue`** & το **`Key`** με την **`newValue`** και επιστρέφει true, αλλιώς false
- **`V replace(Key, newValue)`** - αντικαθιστά το mapping -μόνο αν υπάρχει κάποιο value συνδεδεμένο με το **`Key`**- με την **`newValue`** και επιστρέφει το previous value (αν υπήρχε αλλιώς **`null`** και δεν γίνεται το **`replace()`**)
- **`Set<Map.Entry<K,V>> entrySet()`** - επιστρέφει ένα Set των mappings
- Μέθοδοι του **`Interface Map.Entry<K, V>`** (αναπαριστά ένα entry του **`HashMap<K, V>`**
  - **`getKey()`** – επιστρέφει το Key του Entry
  - **`getValue()`** – Επιστρέφει το value του Entry

Διάσχιση ενός HashMap

- Όπως είδαμε για να ***διασχίσουμε (traverse)*** δηλ. να επισκεφτούμε ένα-ένα τους κόμβους του HashMap μπορούμε να το κάνουμε με τρεις τρόπους:
  - Κλασσική for-each που τρέχει στο **`entrySet()`**
  - Iterator – σε στοιχεία Map.Entry<>
  - Μέθοδος **`forEach()`**

Αναζήτηση map-entry με βάση το κλειδί και remove


```
29          Iterator<Map.Entry<String, String>> it2 = countries.entrySet().iterator();
30          while (it2.hasNext()) {
31              Map.Entry<String, String> entry = it2.next();
32              if (entry.getKey().equals("FR")) {
33                  it2.remove();
34                  System.out.println(entry.getKey() + ":" + entry.getValue() + " deleted");
35              }
36          }
```

- Το remove γίνεται με ασφάλεια μόνο μέσω του Iterator.

Διαγραφή Entry


```
29      Iterator<Map.Entry<String, String>> it2 = countries.entrySet().iterator();
30      while (it2.hasNext()) {
31          Map.Entry<String, String> entry = it2.next();
32          if (entry.getValue().equals("France")) {
33              it2.remove();
34              System.out.println(entry.getKey() + ":" + entry.getValue() + " deleted");
35          }
36      }
```

- Όπως έχουμε ξαναδεί ο μόνος συνεπής τρόπος τροποποίησης ενός collection είναι με Iterator
- Εναλλακτικά διαγράφουμε με Map.remove(Key)

Διαγραφή πολλαπλών entries


```
38      Iterator<Map.Entry<String, String>> it3 = countries.entrySet().iterator();
39      List<String> toBeRemoved = new ArrayList<>();
40      while (it3.hasNext()) {
41          Map.Entry<String, String> entry = it3.next();
42          if (entry.getKey().matches("[A-Z]{2}")) {
43              toBeRemoved.add(entry.getKey());
44          }
45      }
46
47      toBeRemoved.forEach(countries::remove);
48      countries.forEach((k, v) -> System.out.println(k + ":" + v));
```

- Εισάγουμε σε μια λίστα τα κλειδιά που πληρούν το κριτήριο των δύο χαρακτήρων (βλ. matches) και μετά διαγράφουμε

Map Initializers


```java
// JDK 8 Double Braces - Anonymous class plus instances init block
Map<String, Object> criteria = new HashMap<>() {{
    put("price", 10.70);
    put("quantity", 10);
    put("description", "Oranges");
}};

// >= JDK9 - Unmodifiable Map
Map<String, Object> criteria2 =
        Map.of("price", 10.70, "quantity", 10, "description", "Oranges");

// >= JDK9 - Modifiable Map
var criteria3 = new HashMap<String, Object>(
        Map.of("price", 10.70, "quantity", 10, "description", "Oranges")
);
```

Map Streams (1)


```
9   public static void main(String[] args) {
10
11      //populate
12      Map<String, Product> products = new HashMap<>() {{
13          put("ORA", new Product("ORA", "Oranges", 12.5, 19));
14          put("HON", new Product("HON", "Honey", 10.5, 10));
15          put("MIL", new Product("MIL", "Milk", 3.8, 3));
16          put("EGG", new Product("EGG", "Eggs", 1.8, 1));
17      }};
18
19      List<Product> pros =  products.values().stream()
20              .filter(product -> product.getPrice() > 10.2)
21              .collect(Collectors.toList());
22
23      pros.forEach(System.out::println);
24
```

Map Streams (2)


```
9   public static void main(String[] args) {
10
11      //populate
12      Map<String, Product> products = new HashMap<>() {{
13          put("ORA", new Product("ORA", "Oranges", 12.5, 19));
14          put("HON", new Product("HON", "Honey", 10.5, 10));
15          put("MIL", new Product("MIL", "Milk", 3.8, 3));
16          put("EGG", new Product("EGG", "Eggs", 1.8, 1));
17      }};
18
19      int total =  products.values().stream()  Stream<Product>
20              .filter(product -> product.getPrice() > 10.2)
21              .mapToInt(Product::getQuantity)  IntStream
22              .sum();
23
24      System.out.println(total);
25
```

Map Streams (3)


```
9   public static void main(String[] args) {
10
11      //populate
12      Map<String, Product> products = new HashMap<>() {{
13          put("ORA", new Product("ORA", "Oranges", 12.5, 19));
14          put("HON", new Product("HON", "Honey", 10.5, 10));
15          put("MIL", new Product("MIL", "Milk", 3.8, 3));
16          put("EGG", new Product("EGG", "Eggs", 1.8, 1));
17      }};
18
19      int total =  products.values().stream()  Stream<Product>
20              .filter(product -> product.getPrice() > 10.2)
21              .mapToInt(Product::getQuantity)  IntStream
22              .reduce(0, (tot, qua) -> Integer.sum(tot, qua));
23
24      System.out.println(total);
```

Map Streams (4)


```
9   public static void main(String[] args) {
10
11      //populate
12      Map<String, Product> products = new HashMap<>() {{
13          put("ORA", new Product("ORA", "Oranges", 12.5, 19));
14          put("HON", new Product("HON", "Honey", 10.5, 10));
15          put("MIL", new Product("MIL", "Milk", 3.8, 3));
16          put("EGG", new Product("EGG", "Eggs", 1.8, 1));
17      }};
18
19      int total =  products.values().stream()  Stream<Product>
20              .filter(product -> product.getPrice() > 10.2)
21              .mapToInt(Product::getQuantity)  IntStream
22              .reduce(0, Integer::sum);
23
24      System.out.println(total);
```

Map Streams (5)


```
9   public static void main(String[] args) {
10
11      //populate
12      Map<String, Product> products = new HashMap<>() {{
13          put("ORA", new Product("ORA", "Oranges", 12.5, 19));
14          put("HON", new Product("HON", "Honey", 10.5, 10));
15          put("MIL", new Product("MIL", "Milk", 3.8, 3));
16          put("EGG", new Product("EGG", "Eggs", 1.8, 1));
17      }};
18
19
20      // JDK8 Double Braces
21      Map<String, Object> criteria = new HashMap<>() {
22          {
23              put("description", "Eggs");
24          }
25      };
26
27      String eggs = products.values().stream() Stream<Product>
28              .filter(pro -> pro.getDescription().equals(criteria.get("description")))
29              .map(Product::toString) Stream<String>
30              .collect(Collectors.joining());
31
32      System.out.println(eggs);
```

- Αναζήτηση με κριτήρια
- Τα κριτήρια λειτουργούν στο πλαίσιο Predicates

Map Streams (6)


```
11          //populate
12          Map<String, Product> products = new HashMap<>() {{
13              put("ORA", new Product("ORA", "Oranges", 12.5, 19));
14              put("HON", new Product("HON", "Honey", 10.5, 10));
15              put("MIL", new Product("MIL", "Milk", 3.8, 3));
16              put("EGG", new Product("EGG", "Eggs", 8.8, 1));
17          }};
18
19
20          // JDK8 Double Braces
21          Map<String, Object> criteria = new HashMap<>() {
22              {
23                  put("description", "Eggs");
24                  put("price", 8.0);
25              }
26          };
27
28          String eggs = products.values().stream() Stream<Product>
29                  .filter(pro -> pro.getDescription().equals(criteria.get("description"))
30                          && (Double.compare(pro.getPrice(), (double) criteria.get("price"))) > 0)
31                  .map(Product::toString) Stream<String>
32                  .collect(Collectors.joining());
33
34          System.out.println(eggs);
```

Map Streams (7)


```
19          List<Product> products = new ArrayList<>(List.of(
20                  new Product("ORA", "Oranges A", 2.5, 100),
21                  new Product("EGG", "Eggs", 8, 6),
22                  new Product("APP", "Apples", 4.7, 5),
23                  new Product("MIL", "Milk", 1.8, 1),
24                  new Product("ORA", "Oranges B", 12.2, 1),
25                  new Product("ORA", "Oranges C", 17.2, 5)
26          ));
27
28
29          // JDK8 Double Braces
30          Map<String, Object> criteria = new HashMap<>() {
31              {
32                  put("id", "ORA");
33                  put("price", 5.0);
34              }
35          };
36
37          List<Product> upperOranges = products.stream()
38                  .filter(pro -> pro.getId().equals(criteria.get("id"))
39                          && (Double.compare(pro.getPrice(), (double) criteria.get("price"))) > 0)
40                  .collect(Collectors.toList());
41
42          upperOranges.forEach(System.out::println);
```

class TreeMap (1)

- Υλοποίηση Map με ισοζυγισμένο red-Black tree
- Διατηρεί ordering με βάση το natural ordering των κλειδιών ή με βάση Comparator κατά τη δημιουργία
- Σταθερός χρόνος O(logn) για εισαγωγή, διαγραφή, αναζήτηση (put(), remove(), get())

Constructor
`TreeMap()`
`TreeMap(Comparator<? super K> comparator)`
`TreeMap(Map<? extends K,? extends V> m)`
`TreeMap(SortedMap<K,? extends V> m)`

class TreeMap (2)


```
51          SortedMap<String, String> sortedCountries = new TreeMap<>(countries);
52          sortedCountries.forEach((k, v) -> System.out.println(k + "" + v));
```

```
"C:\Program Files\Java\jdk1.8.0_40\
FR, France
GR, Greece
IT, Italy
USA, United States

Process finished with exit code 0
```

- Ταξινομημένη διάσχιση σε αύξουσα σειρά (φυσική κατάταξη) με TreeMap

class LinkedHashMap

- Υλοποίηση με Hash Table και Linked list
- Ordering με βάση την σειρά εισαγωγής των κλειδιών
- Επομένως αν αλλάξει ένα value το ordering διατηρείται

Enumerations

- Τα Enums είναι ένας ορισμός παρόμοιος με τον ορισμό μίας κλάσης, όπου μπορούμε και ορίζουμε σταθερές τιμές
- Οι σταθερές στην πραγματικότητα αντιστοιχούν σε instances της κλάσης, που δημιουργούνται αυτόματα μέσα στην κλάση

AccountType


```java
package gr.aueb.cf.enums;

public enum AccountType {
    DEPOSIT,
    SAVINGS,
    CURRENT
}

```

- Ορίζουμε ένα enum *AccountType* με τρεις σταθερές
- Δηλώσεις του τύπου μπορούμε να κάνουμε:

AccountType type = AccountType.DEPOSIT;

Name / ordinal


```java
package gr.aueb.cf.enums;

public class Main {

    public static void main(String[] args) {
        for (AccountType accountType : AccountType.values()) {
            System.out.println("Name " + accountType.name()
                    + ", ordinal number:  " + accountType.ordinal());
        }
    }
}
```

```
Run:  Main (3)
"C:\Program Files\Amazon Corretto\jdk11.0.10_9\bin\java.exe" "-javaa
Name DEPOSIT, ordinal number:  0
Name SAVINGS, ordinal number:  1
Name CURRENT, ordinal number:  2

Process finished with exit code 0
```

- Τα ordinal numbers είναι αριθμοί που αντιστοιχούν στις σταθερές και ξεκινάνε από το 0

Enum με custom ordinals (1)


```java
package gr.aueb.cf.enums;

public enum AccountType {
    DEPOSIT("DP"),
    SAVINGS("SA"),
    CURRENT("CU");

    private String code;

    AccountType(String code) {
        this.code = code;
    }

    public String getCode() {
        return code;
    }
}
```

- Αν δεν θέλουμε 0, 1, 2, κλπ. ως ordinal numbers αλλά custom ordinals, τότε μπορούμε να ορίσουμε τις σταθερές όπως θα καλούσαμε ένα constructor, το οποίο και πρέπει να ορίσουμε, όπως και το αντίστοιχο πεδίο
- Ο constructor (πρέπει να) είναι private (αν δεν το δώσουμε υπονοείται) μιας και χρησιμοποιείται εσωτερικά από το enum για να δημιουργήσει instances. Δεν μπορούμε δηλαδή να κάνουμε άμεσα instantiate

Enum με custom ordinals (2)

- Τα ordinals 0, 1, 2 κλπ. παραμένουν (μιας και δεν μπορεί η ordinal() να υπερκαλυφθεί, είναι final) αλλά δημιουργούμε custom ordinals με τον τρόπο που είδαμε

Ημέρες εβδομάδας


```java
package gr.aueb.cf.testbed.ch19;

public enum WeekDay {
    SUNDAY("Sunday"),
    MONDAY("Monday"),
    TUESDAY("Tuesday"),
    WEDNESDAY("Wednesday"),
    THURSDAY("Thursday"),
    FRIDAY("Friday"),
    SATURDAY("Saturday");

    private final String day;

    private WeekDay(String day) {
        this.day = day;
    }

    public String getDay() {
        return day;
    }
}
```

```java
package gr.aueb.cf.testbed.ch19;

public class DaysApp {

    public static void main(String[] args) {
        WeekDay weekDays;

        for (WeekDay weekDay : WeekDay.values()) {
            System.out.println(weekDay.getDay());
        }
    }
}
```

Εργασία

- Υλοποιήστε την εργασία/εφαρμογή του Account (του κεφαλαίου 18) ορίζοντας στο DAO Layer ως datasource ένα HashMap με key το id του Account, που είναι Long και value το Account
- Τροποποιήστε ανάλογα την υλοποίηση του Public API