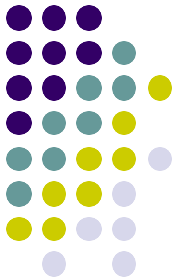
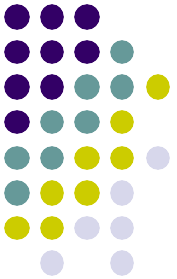
Collections



Ionut

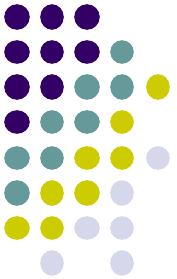
Spalatelu

Outline



- collection vs Collection vs Collections
- Java collections framework
- Operations on collections
- Collection interfaces
- List
- Set: equals() and hashcode()
- Map

collection vs Collection vs Collections

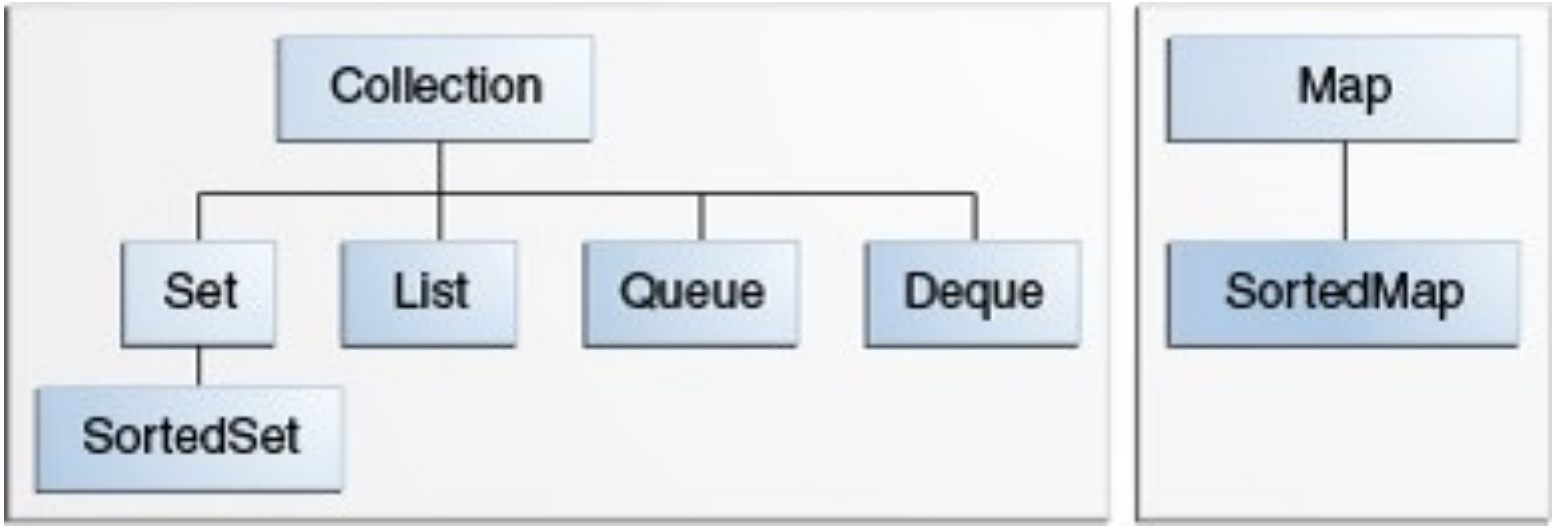


- **collection**(lowercase **c**): the general term for describing a data structure in which objects are stored and iterated over

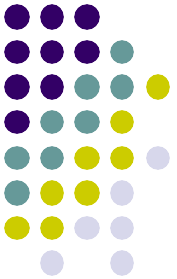
- **Collection** (capital **C**), which is actually the java.util.Collection super-interface from which all other specialized interfaces (Set, List, and Queue) extend

- **Collections** – utility class that holds a pile of static methods implementing various algorithms like: searching, sorting, shuffling etc. Tipically contains algorithm implementations that don't fit well in their associated data structure.

Java collections framework



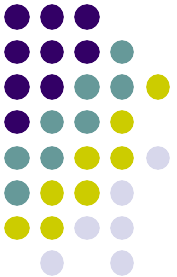
Operations on collections



Operations that can be done on a collection:

- **Add** objects to the collection
- **Retrieve** an object from the collection (without removing it)
- **Remove** objects from the collection
- **Find** out if an object (or group of objects) is in the collection
- **Iterate** through the collection, looking at each element (object) one after another

Collection interfaces



Any collection of elements can be:

- **Ordered**
  - when a collection is ordered, it means we can iterate through the collection in a specific (not-random) order
- **Unordered**
- **Sorted**
  - a sorted collection means that the order in the collection is determined according to some rule or rules, known as the sort order
  - sorting is done based on properties of the objects themselves

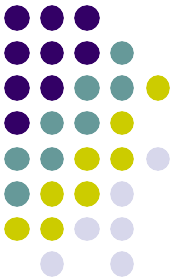
- **Unsorted**

An implementation class can be:

- unsorted and unordered
- ordered but unsorted
- both ordered and sorted

An implementation can NEVER be sorted but unordered!

Collection interfaces



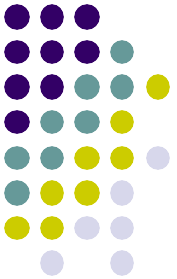
- **List:**
  - ordered and unsorted
  - keeps the insertion order
  - elements have an index (just like an array)
  - duplicates are allowed

- **Set:**
  -unordered and unsorted (there is a special impl. which keeps elements sorted)
  - no duplicates are allowed (unique elements)

- **Queue:**
  - ordered and unsorted
  - elements ordered by their processing order
  - duplicates are allowed

- **Map:**
  - contains key-value pairs
  - unordered and unsorted (for keys)
  - no duplicates are allowed (for keys <-> form a Set)

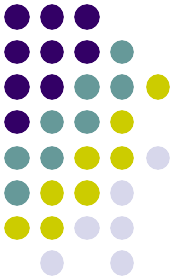
Collection



Collection interface contains some general operations for collections:

- basic operations: **add(E), remove(Object), contains(Object), isEmpty(), size()**
- bulk operations: **addAll(Collection), removeAll(Collection), containsAll(Collection), clear()**

List



Characteristics:

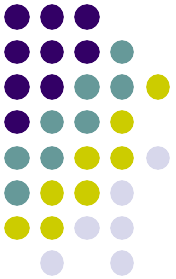
- **Positional access** — manipulates elements based on their index in the list. This includes methods such as **get, set, add, addAll, and remove**.

- **Search** — searches for a specified object in the list and returns its index. Search methods include **indexOf and lastIndexOf.**

- **Range-view** — The **sublist** method performs arbitrary *range operations* on the list.

- **Implementations**: ArrayList, LinkedList, Vector(deprecated)

List



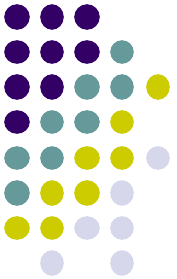
Performance for different implementations:

- ArrayList:
  - get(int index) is *O(1)*
  - add(E element) is *O(n)*
  - add(int index, E element) is *O(n)*
  - remove(int index) is *O(n)*
  - Iterator.remove() is *O(n)*

- LinkedList
  - get(int index) is O(n) but O(1) for the ends with **getFirst()** and **getLast()**
  - add(int index, E element) is *O(n)* but *O(1)* when use **addFirst()** and **addLast()**
  - remove(int index) is *O(n)* but *O(1)* when use **removeFirst()** and **removeLast()**
  - Iterator.remove() is *O(1)*

- Vector similar to ArrayList but with syncronized acces, which causes an important performance degradation

Set

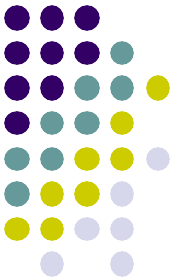


- Set's main asset is uniqueness. For this, all of his implementations rely on two of the Object's methods: equals() and hashcode().

- equals() and hashcode() contract: **If two objects are considered equal, their hashcodes must also be equal! Reverse is not mandatory! a.equals(b) ==> a.hashcode() == b.hashcode()**

- The contract between the 2 methods it’s easily followed if they are always **overriden together** and using the same subset of fields for that class

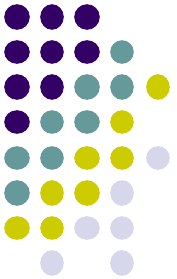
Set



The **equals()** properties to be considered when is overriden:

- **reflexive** - for any reference value x, x.equals(x) should return true
- **symmetric** - for any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true
- **transitive** - for any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must return true
- **consistent** - for any reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified
- for any non-null reference value x, x.equals(null) should return false

Set



The **hashcode()** properties to be considered when is overriden:

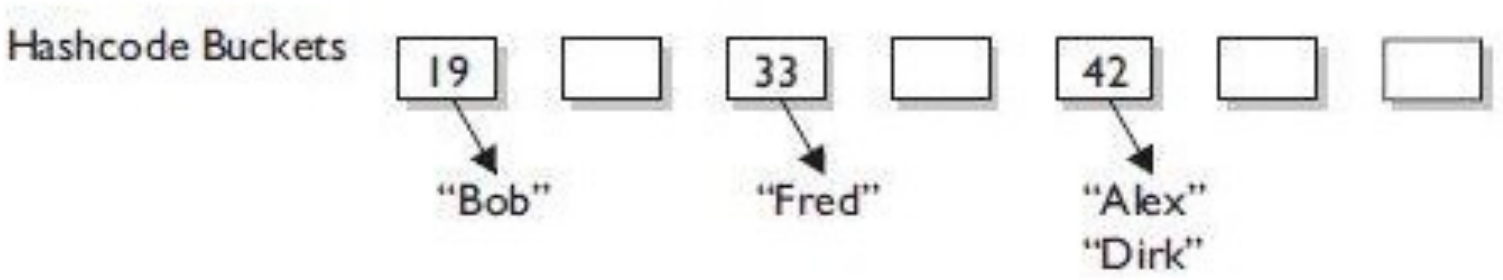
- whenever it is invoked on the same object more than once, it must consistently return the same integer, provided no information used in equals() comparisons on the object is modified

- it is NOT required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode() method on each of the two objects must produce distinct integer results;

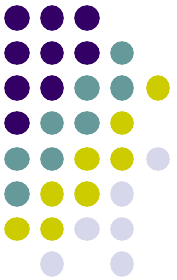
Set

```
Set<String> fellows = new HashSet<>();
fellows.add("Alex");
fellows.add("Bob");
fellows.add("Dirk");
fellows.add("Fred");
```

Key	Hashcode Algorithm	Hashcode
Alex	A(1) + L(12) + E(5) + X(24)	= 42
Bob	B(2) + O(15) + B(2)	= 19
Dirk	D(4) + I(9) + R(18) + K(11)	= 42
Fred	F(6) + R(18) + E(5) + D(4)	= 33



Comparator vs Comparable



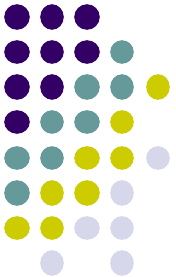
Comparable:This interface imposes a total ordering on the objects of the class that **implements it**. This ordering is referred to as the class's *natural ordering*, and the class's **compareTo** method is referred to as its *natural comparison method*.

Contract: Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

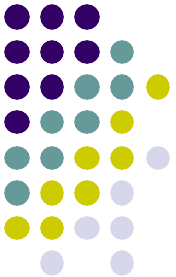
Comparator: A comparison interface, which imposes a total ordering on some collection of objects. Comparators are **separate objects**, that can be passed to a sort method (such as Collections.sort or Arrays.sort) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as sorted collections), or to provide an ordering for collections of objects that don't have a natural ordering (don't implement Comparable).

Contract: Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

Comparator vs Comparable



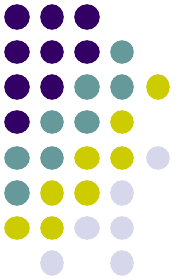
java.lang.Comparable	java.util.Comparator
int objOne.**compareTo**(objTwo)	int **compare**(objOne, objTwo)
Returns: **negative** if objOne < objTwo **zero** if objOne == objTwo **positive** if objOne > objTwo	Same as Comparable
You must modify the class whose instances you want to sort	You build a class separate from the class whose instances you want to sort
Only **one** sort sequence can be created	**Many** sort sequences can be created
Implemented frequently in the API by: String, Wrapper classes, Date.	Meant to be implemented to sort instances of third-party classes



Comparator vs Comparable

Class	Natural Ordering
Byte	Signed numerical
Character	Unsigned numerical
Long	Signed numerical
Integer	Signed numerical
Short	Signed numerical
Double	Signed numerical
Float	Signed numerical
BigInteger	Signed numerical
BigDecimal	Signed numerical
Boolean	Boolean.FALSE < Boolean.TRUE
File	System-dependent, usually on path name
String	Lexicographic
Date	Chronological

Set



Performance for different implementations:

- HashSet:
  - add, remove, contains: O(1)
- TreeSet
  - add, remove, contains: O(log n)
- LinkedHashSet
  - add, remove, contains: O(1) but with a bigger constant

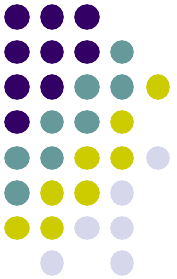
Queue

A Queue is a collection for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, removal, and inspection operations.

Queue interface structure

Type of Operation	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

Map



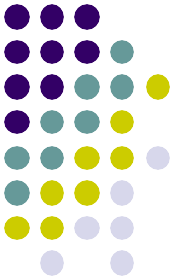
The three general-purpose Map implementations are **HashMap**, **TreeMap** and **LinkedHashMap**.

- If you want key-sorted Collection-view iteration, use **TreeMap**;
- if you want maximum speed and don't care about iteration order, use **HashMap**;
- if you want near-HashMap performance and insertion-order iteration, use **LinkedHashMap**.

  In all of the three cases the situation for **Map** is analogous to Set having in mind that you are always operating on the keys (the key Set).

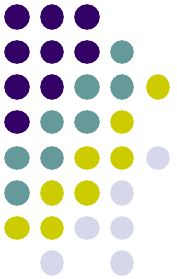
  Likewise, everything else in the Set implementations section also applies to Map implementations.

Collections summary



Class	Map	Set	List	Ordered	Sorted
`HashMap`	x			No	No
`Hashtable`	x			No	No
`TreeMap`	x			Sorted	By *natural order* or custom comparison rules
`LinkedHashMap`	x			By insertion order or last access order	No
`HashSet`		x		No	No
`TreeSet`		x		Sorted	By *natural order* or custom comparison rules
`LinkedHashSet`		x		By insertion order	No
`ArrayList`			x	By index	No
`Vector`			x	By index	No
`LinkedList`			x	By index	No

Let's choose a collection



- Choose the right type of collection based on the need:
  - if we need access based on the index, we might want to use .....?
  - if we have to iterate over the key-value collection in order of insertion, we need to use ....?
  - if we don't want duplicates, we should use a ....?
  - if you just you pull a sorted collection of Employees from database you should use a …?

- Write programs in terms of interfaces not implementations, it allows to change the implementation easily at a later point of time

Questions