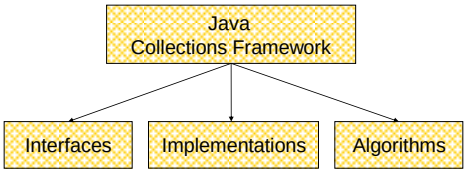
תוכנה 1

תרגול 6 – מבני נתונים גנריים

הדס צור ואסף זריצקי

Java Collections Framework

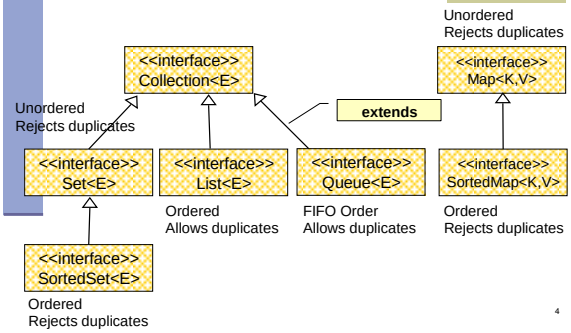
- **Collection**: a group of elements
- Interface Based Design:



Online Resources

- Java 6 API Specification:
  http://java.sun.com/javase/6/docs/api/
  - The Collections framework in java.util
- Sun Tutorial:
  http://java.sun.com/docs/books/tutorial/collections/

Collection Interfaces



A Simple Example

```
Collection<String> stringCollection = ...
Collection<Integer> integerCollection = ...

stringCollectio
integerCollect
integerCollect

stringCollection.add(7);
integerCollection.add("world");
stringCollection = integerCollection;
```

- מצביעים ל Collection של מחרוזות ושל מספרים
- Collection אינו מחזיק טיפוסים פרימיטיביים, לכן נשתמש ב Integer, Double, Float וכדומה
- נראה בהמשך אילו מחלקות מממשות מנשק זה

A Simple Example

```
Collection<String> stringCollection = ...
Collection<Integer> integerCollection = ...

stringCollection.add("Hello");
integerCollection.add(5);
integerCollection.add(new Integer(6));

stringCollection.add(7);
integerCollection.add("world");
stringCollection = integerCollection;
```

Best Practice <with generics>

- Specify an element type only when a collection is instantiated:
  - `Set<String> s = new HashSet<String>();`

    Interface — Implementation

    Works, but...
  - `public void foo(HashSet<String> s){…}`
  - `public void foo(Set<String> s) {…}`
  - `s.add()` invokes `HashSet.add()`

    Better!

    polymorphism

8

A Simple Example

```
Collection<String> stringCollection = ...
Collection<Integer> integerCollection = ...

stringCollection.add("Hello");
integerCollection.add(5);
integerCollection.add(new Integer(6));

stringCollection.add(7);
integerCollection.add("world");
stringCollection = integerCollection;
```

7

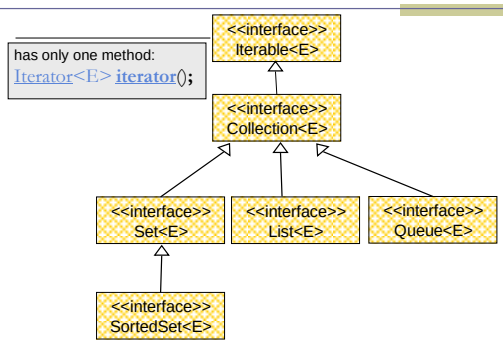
The Iterator Interface

- Provide a way to access the elements of a collection sequentially without exposing the underlying representation
- Methods:
  - **hasNext()** - Returns true if there are more elements
  - **next()** - Returns the next element
  - **remove()** - Removes the last element returned by the iterator (optional operation)

Command and Query

10

Collection extends Iterable



9

Collection Implementations

- Class Name Convention: <Data structure> <Interface>

General Purpose Implementations		Data Structures			
		Hash Table	Resizable Array	Balanced Tree	Linked List
Interfaces	Set	HashSet		TreeSet (SortedSet)	
	Queue		ArrayDeque		LinkedList
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap (SortedMap)	

12

Iterating over a Collection

- Explicitly using an Iterator

```
for (Iterator<String> iter = stringCollection.iterator();
     iter.hasNext(); ) {
    System.out.println(iter.next());
}
```

- Using foreach synatx

```
for (String str : stringCollection) {
    System.out.println(str);
}
```

11

List Example

```
List<Integer> list = new ArrayList<Integer>();
list.add(3);
list.add(1);
list.add(new Integer(1));
list.add(new Integer(6));
list.remove(list.size()-1);
System.out.println(list);
```

Output:
[3, 1, 1]

Interface

Implementation

List holds Integer references (auto-boxing)

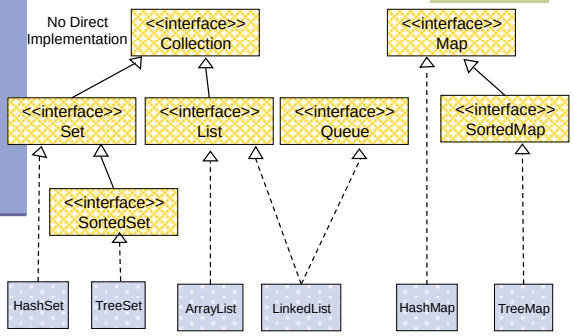
List allows duplicates

Invokes List.toString()

remove() can get *index* or *reference* as argument

Insertion order is kept

General Purpose Implementations



Map Example

```
Map<String,String> map = new HashMap<String,String>();
map.put("Dan", "03-9516743");
map.put("Rita", "09-5076452");
map.put("Leo", "08-5530098");
map.put("Rita", "06-8201124");
System.out.println(map);
```

Output:
{Leo=08-5530098, Dan=03-9516743, Rita=06-8201124}

No duplicates

Unordered

Keys (names)	Values (phone numbers)
Dan	03-9516743
Rita	06-8201124
Leo	08-5530098

Set Example

```
Set<Integer> set = new HashSet<Integer>();
set.add(3);
set.add(1);
set.add(new Integer(1));
set.add(new Integer(6));
set.remove(6);
System.out.println(set);
```

A set does not allow duplicates.
It **does not** contain:
- two references to the same object
- two references to null
- references to two objects a and b such that a.equals(b)

remove() can get only *reference* as argument

Output: `[1, 3] or [3, 1]`

Insertion order is not guaranteed

Map Collection Views

Three views of a `Map<K,V>` as a collection

`keySet` — Set<K>

`values` — Collection<V>

`entrySet` — Set<Map.Entry<K,V>>

The `Set` of key-value pairs (implement `Map.Entry`)

SortedMap Example

```
SortedMap <String,String>map = new TreeMap<String,String> ();
map.put("Dan", "03-9516743");
map.put("Rita", "09-5076452");
map.put("Leo", "08-5530098");
map.put("Rita", "06-8201124");
System.out.println(map);
```

Output:
{Dan=03-9516743, Leo=08-5530098, Rita=06-8201124}

lexicographic order

Keys (names)	Values (phone numbers)
Dan	03-9516743
Rita	06-8201124
Leo	08-5530098

Iterating Over the Keys of a Map

```
Map<String,String> map = new  HashMap<String,String> ();
map.put("Dan", "03-9516743");
map.put("Rita", "09-5076452");
map.put("Leo", "08-5530098");
map.put("Rita", "06-8201124");

for (String key : map.keySet()) {
    System.out.println(key);
}
```

Output:

```
Leo
Dan
Rita
```


Iterating Over the Keys of a Map

```
Map<String,String> map = new  HashMap<String,String> ();
map.put("Dan", "03-9516743");
map.put("Rita", "09-5076452");
map.put("Leo", "08-5530098");
map.put("Rita", "06-8201124");

for (Iterator<String> iter= map.keySet().iterator(); iter.hasNext(); ) {
    System.out.println(iter.next());
}
```

Output:

```
Leo
Dan
Rita
```


Iterating Over the Key-Value Pairs of a Map

```
Map<String,String> map = new HashMap<String,String> ();
map.put("Dan", "03-9516743");
map.put("Rita", "09-5076452");
map.put("Leo", "08-5530098");
map.put("Rita", "06-8201124");

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

Output:

```
Leo: 08-5530098
Dan: 03-9516743
Rita: 06-8201124
```


Iterating Over the Key-Value Pairs of a Map

```
Map<String,String> map = new HashMap<String,String>();
map.put("Dan", "03-9516743");
map.put("Rita", "09-5076452");
map.put("Leo", "08-5530098");
map.put("Rita", "06-8201124");

for (Iterator<Map.Entry<String,String>> iter= map.entrySet().iterator();
    iter.hasNext(); ) {
    Map.Entry<String,String> entry = iter.next();
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

Output:

```
Leo: 08-5530098
Dan: 03-9516743
Rita: 06-8201124
```


Sorting

bridge between array-based and collection-based APIs

```
public class SortExample {
    public static void main(String args[]) {
        List<String> list = Arrays.asList(args);

        System.out.println("Before sort: " + list);
        Collections.sort(list);
        System.out.println("After sort: " + list);
    }
}
```

- Sorts the list according to the *natural ordering* of its elements
- All elements in the list must implement the Comparable interface

Collection Algorithms

- Defined in the Collections class
  - Similar to Arrays
- Main algorithms:
  - sort
  - binarySearch
  - reverse
  - shuffle
  - min / max

Sorting User Defined Class

- Now we can use Collections.sort

```
public static void main(String[] args) {
    List<Student> students = new ArrayList<Student>();
    students.add(new Student(100, "Hadas"));
    students.add(new Student(50, "Lior"));
    students.add(new Student(1, "Mati"));
    students.add(new Student(200, "Assaf"));

    System.out.println("Before sort: " + students);
    Collections.sort(students);
    System.out.println("After sort: " + students);
}
```

Before sort: [<Hadas, 100>, <Lior, 50>, <Mati, 1>, <Assaf, 200>]
After sort: [<Mati, 1>, <Lior, 50>, <Hadas, 100>, <Assaf, 200>]

Sorting User Defined Class

- We need to implement the C*omparable* interface

Can compare two Student objects

```
public class Student implements Comparable<Student> {
    private int id;
    private String name;

    @Override
    public int compareTo(Student o) {
        return id - o.id;
    }

    // more methods ...
}
```

id based comparison

Unnatural Sort

- Sort objects that don't implement *Comparable*
- Define a different ordering (e.g. students by name)
- Provide the *sort* algorithm a comparison function
  - implement ***Comparator<T>***

```
public class StudentNameComparator
   implements Comparator<Student> {

    public int compare(Student s1, Student s2) {
        return s1.getName().compareTo(s2.getName());
    }
}
```