Advanced Object-Oriented Design
Lecture 2

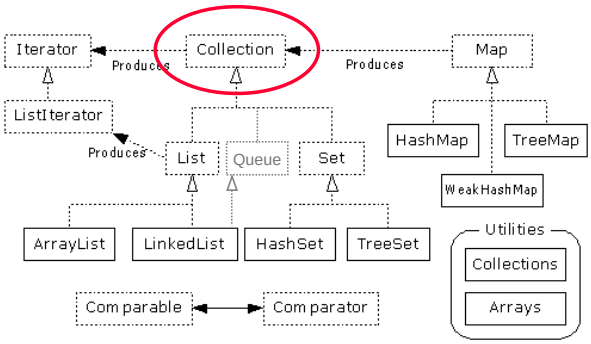
Java 2 Collections

Bartosz Walter
<Bartek.Walter@man.poznan.pl>

Agenda

- **Basic interfaces: Collection, Set, List, Queue, Map**
- **Iterators**
- **Utility classes for Collections and Arrays**
- **Wrapper implementations**
- **Facilities for comparing objects**

Java 2 Collections



java.util.Collection

- Manages a group of ***elements***
- Least common denominator that all collections implement
- Imposes no constraints on the elements
- No direct implementation
- Mutating methods throw *UnsupportedOperationException*

General constraints on implementations

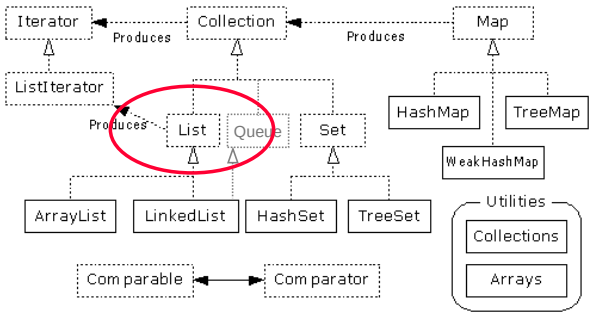
- Two obligatory constructors:
  - parameterless `Collection()`
  - Collection-based `Collection(Collection source)`

java.util.Collection

Groups of methods:

- basic operations – adding, removing, measuring
  - `boolean add(Object o)`
  - `boolean remove(Object o)`
  - `boolean contains(Object o)`
  - `boolean isEmpty()`
  - `Iterator iterator()`
- bulk operations – operations on groups of elements
  - `boolean addAll(Collection coll)`
  - `boolean removeAll(Collection coll)`
  - `boolean containsAll(Collection coll)`
  - `boolean retainAll(Collection coll)`
  - `void clear()`
- array operations – conversion to arrays
  - `Object[] toArray()`
  - `Object[] toArray(Object[])`

Java 2 Collections



java.util.List

- Subinterface of Collection
- Manages a group of ***ordered elements***
- Usually allows for storing duplicates
- Direct implementations: `ArrayList, Vector, LinkedList`

Specification

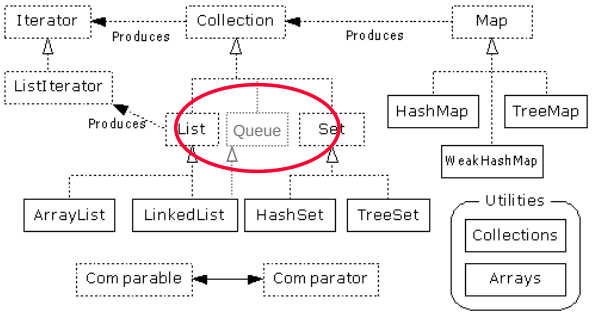
- Positional Access: manipulate elements based on their numerical position in the list
- Search: search for a specified object in the list and return its numerical position
- List Iteration: extend *Iterator* semantics to take advantage of the list's sequential nature
- Range-view: perform arbitrary *range operations* on the list

java.util.List

Groups of methods:

- positional access
  - `Object get(int index)`
  - `Object set(int index, Object o)`
  - `Object add(int index, Object o)`
  - `Object remove(int index)`
- search
  - `int indexOf(Object o)`
  - `int lastIndexOf(Object o)`
- iteration
  - `ListIterator listIterator()`
  - `ListIterator listIterator(int index)`
- range view
  - `List subList(int from, int to)`

Java 2 Collections



java.util.Queue

- Subinterface of Collection
- Manages an ordered group of elements prior to processing, composed of *head* and *tail*
- Direct implementations: `LinkedList, PriorityQueue`

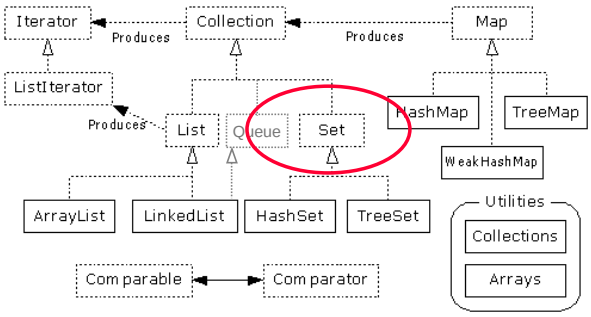
```
public class Countdown {
  public static void main(String[] args) {
    int number = Integer.parseInt(args[0]);
    Queue<Integer> queue = new LinkedList<Integer>();
    for (int i = number; i >= 0; i--)
      queue.add(i);
    while(!queue.isEmpty())
      System.out.println(queue.remove());
  }
}
```

java.util.Queue

Groups of methods:

- adding an element
  - `boolean add(Object o)`
  - `boolean offer(Object o)`
- extracting an element
  - `Object remove()`
  - `Object poll()`
- inspection
  - `Object element()`
  - `Object peek()`

Java 2 Collections

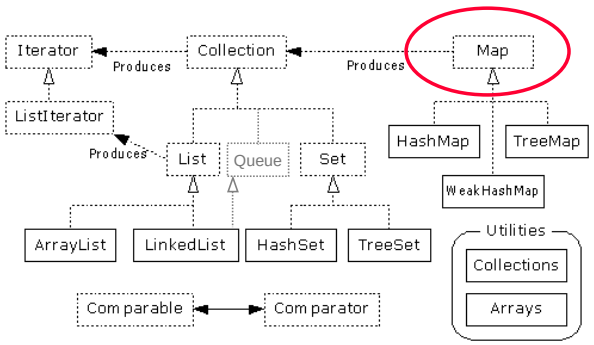


java.util.Set

- Subinterface of Collection, but **no new methods** (!)
- Manages a group of **unique elements**
- Constructors ensure to produce a valid set
- Direct implementations: `HashSet, TreeSet`

```
public class FindDups {
    public static void main(String args[]) {
        Set s = new HashSet();
        for (int i = 0; i < args.length; i++)
            if (!s.add(args[i]))
                System.out.println("Duplicate");
    }
}
```

Java 2 Collections



java.util.Map

- Stores pairs, not single objects
- Maps ***keys*** to ***values***
- Provides Collection-oriented views for keys, values and pairs
- Cannot contain duplicate keys
- Implementations: `HashMap, TreeMap, SortedMap`

Specification

- Two obligatory constructors:
  - parameterless `Map()`
  - Map-based `Map(Map source)`

java.util.Map

Groups of methods:

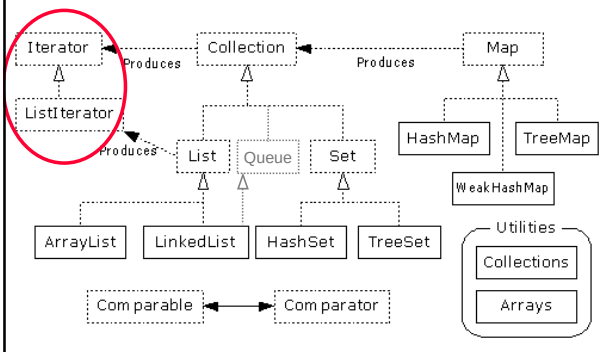
- basic operations
  - `Object put(Object key, Object value)`
  - `Object get(Object key)`
  - `Object remove(Object key)`
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object value)`
  - `boolean isEmpty()`
  - `int size()`
- bulk operations
  - `void putAll(Map map)`
  - `void clear()`
- Collection views
  - `Set keySet()`
  - `Collection values()`
  - `Set entrySet()`

Example

```
public class Freq {
   private static final Integer ONE = new Integer(1);
   public static void main(String args[]) {
      Map m = new TreeMap();
      for (int i=0; i<args.length; i++) {
         Integer freq = (Integer) m.get(args[i]);
         m.put(args[i], (freq == null ? ONE :
            new Integer(freq.intValue() + 1)));
      }
      System.out.println(m.size() + " distinct words:");
      System.out.println(m);
   }
}
```

```
> java Freq if it is to be it is up to me to delegate
8 distinct words:
{to=3, me=1, delegate=1, it=2, is=2, if=1, be=1, up=1}
```

Java 2 Collections



java.util.Iterator

- Generates a serie of elements, one at a time
- Successive calls return successive elements
- Replacement for `java.util.Enumeration`
- No publicly available implementation
- No public constructor

```
boolean hasNext()
Object next()
void remove()
```

Example

```
public class Freq {
   public static void main(String args[]) {
      Collection coll = new ArrayList();
      for (int i=0; i<args.length; i++) {
         coll.add(args[i]);
      }
      for (Iterator iter = coll.iterator(); iter.hasNext();)
         String elem = (String) iter.next();
         System.out.print(elem + "; ");
      }
   }
}
```

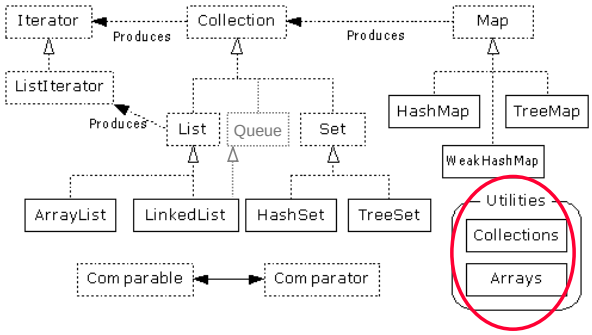
```
> java Freq if it is to be it is up to me to delegate
{if; it; is; be; up; is; me; delegate ...}
```

java.util.ListIterator

- Subinterface of `java.util.Iterator`
- Works in either direction
- Provides convenience methods for operations on the underlying list
- No publicly available implementation
- No public constructor

```
int nextIndex()
void add(Object o)
void set(Object o)
boolean hasPrevious()
Object previous()
int previousIndex()
```

Java 2 Collections



Utility classes: java.util.Collections

Polymorphic implementations of multiple algorithms:

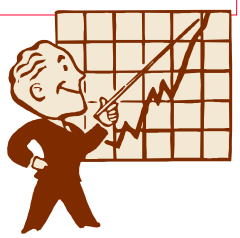
- Binary search for a specific object
- Algebraic operations for Collections
- Sorting arbitrary List (*)
- Copying Lists
- Filling Collections with Objects
- Finding *max* and *min* elements (*)
- Reversing a List
- Shuffling a List
- Swapping two elements

Utility classes: java.util.Collections

```
int binarySearch(List list, Object o)
int binarySearch(List list, Object o, Comparator comp)
void copy(List from, List to)
void fill(List list, Object o)
Object max/min(Collection coll)
Object max/min(Collection coll, Comparator comp)
List nCopies(int no, Object o)
void reverse(List list)
void shuffle(List list)
void sort(List list)
void sort(List list, Comparator comp)
void swap(List list, int idx1, int idx2)
int frequency(Collection coll, Object o)
boolean disjoint(Collection coll1, Collection coll2)
Comparator reverseOrder(Comparator comp)
```

Task

Suggest an implementation of multiset.
A multiset (sometimes also called a bag) differs from a set in that each member has a multiplicity, which is a natural number indicating *how many memberships* it has in the multiset (*Wikipedia*).



Utility classes: java.util.Arrays

- Conversion to a List
- Binary searching for specific object
- Equality of two arrays (deep and shallow)
- *hashCode()* and *toString()* for arrays
- Filling Arrays with Objects
- Sorting Arrays

```
List asList(Object[] objects)
int binarySearch(type[] arr, type o)
int binarySearch(Object[] arr, Object o,
Comparator comp)
boolean equals(type[] arr1, type[] arr2)
void fill(type[] arr, type o)
void sort(type[] arr, Comparator comp)
```

Wrapping objects: Immutables

- Make a Collection immutable
- Block any mutating methods
- No publicly available implementation
- No public constructor

```
Collection unmodifiableCollection(Collection
coll)
Set unmodifiableSet(Set set)
SortedSet unmodifiableSortedSet(SortedSet sset)
List unmodifiableList(List list)
Map unmodifiableMap(Map map)
```

Wrapping objects: Synchronization

- Make Collection synchronized
- Return a thread-safe wrapper for a specified collection
- No publicly available implementation
- No public constructor

```
Collection synchronizedCollection(Collection
coll)
Set synchronizedSet(Set set)
List synchronizedList(List list)
Map synchronizedMap(Map map)
```

Wrapping objects: Type-checking

- Return a type-safe (with respect to its elements) collection

```
Collection checkedCollection(Collection coll)
List checkedList(List list)
Set checkedSet(Set set)
Map checkedMap(Map map)
```

```
Collection<String> c = new HashSet<String>();

Collection<String> c =
    Collections.checkedCollection(
        new HashSet<String>(), String.class);
```

Wrapping objects: Singletons

- Return a collection of one specific element
- The collection is immutable

```
Set singleton(Object o)
List singletonList(List list)
Map singletonMap(Map map)
```

Remove all instances of element e

```
c.removeAll(Collections.singleton(e));
```

Remove all Lawyers from the map:

```
profession.values().removeAll(
    Collections.singleton(LAWYER));
```

Wrapping objects: Empties

- Return an empty collection
- The returned collection is immutable

```
List EMPTY_LIST
Set EMPTY_SET
Collection EMPTY_COLLECTION
Map EMPTY_MAP
```

Comparing objects: java.lang.Comparable

- Sorting, searching requires a method of comparing objects
- Comparison can be extracted and plugged as a parameter
- Comparable objects can be compared to other objects

Sort a List of people by birthday:

```
Collections.sort(people);
```

```
public class Person implements Comparable {
   private String givenName, familyName;
   // ...
   public int compareTo(Object obj) { }
   public boolean equals(Object obj) { }
   public int hashCode() {}
}
```

Comparing objects: java.lang.Comparable

```
public class Person
implements Comparable {
   private String givenName, familyName;

   public String getGivenName() {
      return givenName;
   }

   public int compareTo (Object obj) {
      String name = ((Person) obj).getGivenName();

      return (givenName.compareTo(name));
   }
}
```

Comparing objects: java.util.Comparator

- Sorting, searching requires a method of comparing objects
- The comparison can be extracted and plugged as a param
- Comparators allow for comparing two objects

Sort a List of people by birthday:

```
Collections.sort(people, comparator);
```

```
public class Name implements Comparator {
   public int compare (Object obj1, Object obj2)
   public boolean equals(Object obj)
}
```

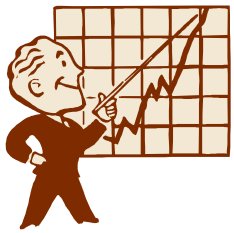
Comparing objects: java.util.Comparator

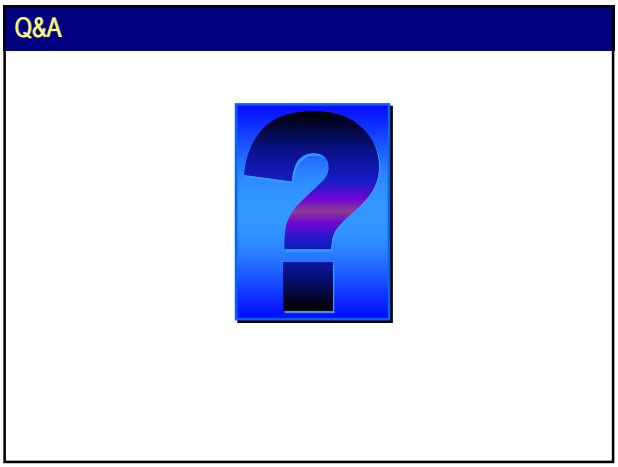
```
public class RealComplexComparator
implements Comparator {

   public int compare (Object obj1, Object obj2) {
      double re1 = ((Complex) obj1).getReal();
      double re2 = ((Complex) obj2).getReal();

      return (re1 > re2? 1: (re1 == re2 ? 0: -1);
   }
}
```

Task

Implement sorting a list of *Person* objects with respect to two different criteria:

- **using *Comparable* interface**
- **using *Comparator*s**




Q&A