Praktische Softwaretechnologie

Lecture 8.

Károly Bósa
(Karoly.Bosa@jku.at)

Research Institute for Symbolic Computation
(RISC)

Arrays vs. Collections

Karoly.Bosa@jku.at

- **Arrays** are defined to be **fixed-size** collections of the same datatype They are the **only** collection that **supports storing primitive data types**.

- A ***collection*** — sometimes called a container — is simply an object that groups multiple elements into a single unit. It **never** has a **pre-defined size**.

What Is a Collections Framework?

Karoly.Bosa@jku.at

All collections frameworks contain the following:

- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation.

- **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.

- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces (see class *Collections*).

An example for a similar structure in C++ is the *Standard Template Library (STL)*

Advantages of the Collections

Karoly.Bosa@jku.at

- **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level data structures.

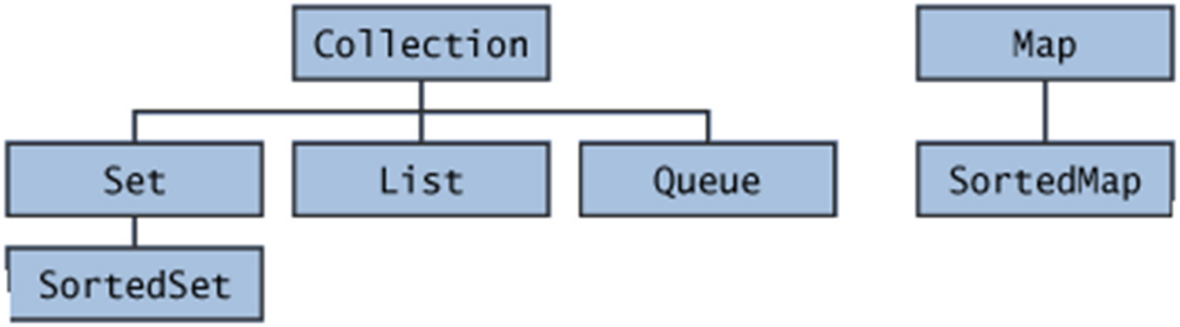
- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms.

- **Allows interoperability:** Different implementations use the same interfaces

- **software reusablity:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

The core collection interfaces

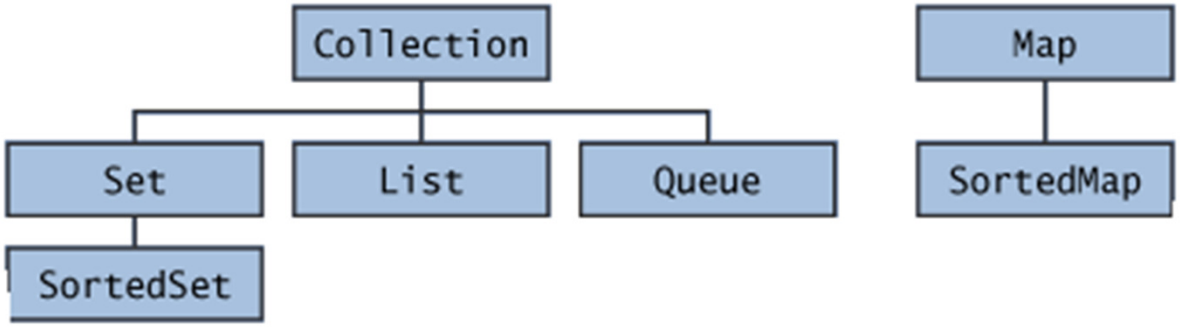
Karoly.Bosa@jku.at



- **Collection** — the root of the collection hierarchy. A collection represents a group of objects known as its elements. **The Java platform doesn't provide any direct implementations of this interface.**
- **Set** — a collection that cannot contain duplicate elements.
- **List** — an ordered collection (sometimes called a *sequence*). Lists can contain duplicate elements. (dynamically resizable array).
- **Queue** — a collection used to hold multiple elements prior to processing. Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner.
- **Map** — an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value (abstraction of functions).

The core collection interfaces

Karoly.Bosa@jku.at



- Note that all the core collection interfaces are generic.

  public interface Collection<E>...

  public interface Map<K, V> …

- Java provides different kind of **implementations** for these interfaces (**usually more than one** for each), except the Collection interface.

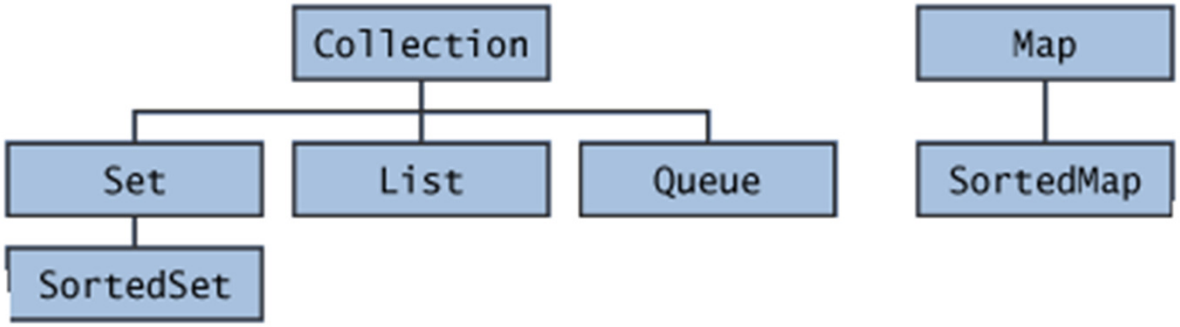
- **Conversion between collection objects via their constructors, e.g.:**

  List<String> list = new ArrayList<String>(c);

  Where **ArrayList** is an implementation of List interface, whose one of constructors expects any kind of object as arguments which implements the collection interface(this means *c* can be TreeSet, LinkedList, etc).

The core collection interfaces

Karoly.Bosa@jku.at



- Note that all the core collection interfaces are generic.

  public interface Collection<E>...

  public interface Map<K, V> …

- Java provides different kind of **implementations** for these interfaces (**usually more than one** for each), except the Collection interface.

- **Conversion between collection objects via their constructors, e.g.:**

  List<String> list = new ArrayList<String>(c);

  Where **ArrayList** is an implementation of List interface, whose one of constructors expects any kind of object as arguments which implements the collection interface(this means *c* can be TreeSet, LinkedList, etc).

The Collection Interface

Karoly.Bosa@jku.at

- A Collection represents a group of objects
- Java platform doesn't provide any direct implementations of this interface.
- It is used to pass around collections of objects where maximum generality is desired.

The Collection Interface

Karoly.Bosa@jku.at

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

        // Basic operations
        int size();
        boolean isEmpty();
        boolean contains(Object element);
        boolean add(E element);
        boolean remove(Object element);
        Iterator<E> iterator();

        // Bulk operations
        boolean containsAll(Collection<?> c);
        boolean addAll(Collection<?> c);
        boolean removeAll(Collection<?> c);
        boolean retainAll(Collection<?> c);
         void clear();

         // Array operations
        Object[] toArray();
        <T> T[] toArray(T[] a);

}
```

Ranging Over Collections

Karoly.Bosa@jku.at

There are two ways to traverse collections:

- **For-each construct** (it cannot modify collections):

```
for (Object o : collection)
    System.out.println(o);
```

- **Iterators** is an object that enables you to traverse through a collection and to remove elements from the collection selectively. The following is the Iterator interface:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

How to Use foreach – Sorting Arguments

Karoly.Bosa@jku.at

Returning with the words of the command line (without duplication)

```
public static void main(String[] args) {
    SortedSet<String> words = new TreeSet<String>();
    for (String s : args) {
        words.add(s);
    }
    for (String s : words) {
        System.out.println(s);
    }
}
```

How to Use Iterators

Karoly.Bosa@jku.at

You get an iterator for a collection by calling its **iterator method**:

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
            if (!cond(it.next())) it.remove();
}
```

Bulk Operations of Collections

Karoly.Bosa@jku.at

- **containsAll** — returns true if the target Collection contains all of the elements in the specified Collection.
- **addAll** — adds all of the elements in the specified Collection to the target Collection.
- **removeAll** — removes from the target Collection all of its elements that are also contained in the specified Collection.
- **retainAll** — removes from the target Collection all its elements that are *not* also contained in the specified Collection. That is, it retains only those elements in the target Collection that are also contained in the specified Collection.
- **clear** — removes all elements from the Collection.

Simple Examples:

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

toArray Operations

Karoly.Bosa@jku.at

- Converting a Collection to an Array
- Usually employed in the case of older APIs that expect arrays on input.
- Example:

```
Object[] a = c.toArray();
```

Or if we suppose that c is known to contain only strings:

```
String[] a = c.toArray(new String[0]);
```

The Set Interface

Karoly.Bosa@jku.at

- A **Set** is a **Collection** that cannot contain duplicate elements.
- It models the mathematical set abstraction.
- The Set interface contains ***only*** **methods inherited** from Collection and adds the restriction that **duplicate elements are prohibited**.
- **Two Set instances are equal if they contain the same elements independently their implementations**

The Set Interface

Karoly.Bosa@jku.at

```
public interface Set<E> extends Collection<E> {
        // Basic operations
        int size();
        boolean isEmpty();
        boolean contains(Object element);
        boolean add(E element);
        boolean remove(Object element);
        Iterator<E> iterator();
        // Bulk operations
        boolean containsAll(Collection<?> c);
        boolean addAll(Collection<? extends E> c);
        boolean removeAll(Collection<?> c);
        boolean retainAll(Collection<?> c);
         void clear();
         // Array operations
        Object[] toArray();
        <T> T[] toArray(T[] a);
}
```

It contains ***only*** **methods inherited** from Collection.

Set Implementation

Karoly.Bosa@jku.at

- **HashSet** stores its elements in a hash table, it is the best-performing implementation; however it makes no guarantees concerning the order of the element.

- **TreeSet** stores its elements in a red-black tree(a kind of **self-balanced binary search tree**), orders its elements based on their values; it is substantially slower than HashSet.

- **LinkedHashSet** is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (**insertion-order**).

Each implementation has a no argument (**HashSet(), TreeSet(), LinkedHashSet()**) constructor and another constructor that expects another Collection object as an argument (**conversion constructors**):

```
Set<Type> s = new HashSet<Type>(c);
```

Example: Finding Duplicates


```
import java.util.*;
public class FindDups {
        public static void main(String[] args) {
            Set<String> s = new HashSet<String>();
            for (String a : args)
                if (!s.add(a))
                    System.out.println("Duplicate detected: " + a);
            System.out.println(s.size() + " distinct words: " + s);    }
}
```

The program takes the words in its argument list and prints out any duplicate words, the number of distinct words, and a list of the words with duplicates eliminated.

Example: Finding Duplicates Output

Now run the program:

java FindDups i came i saw i left

The following output is produced:

Duplicate detected: i
Duplicate detected: i
4 distinct words: [i, left, saw, came]

Example: Finding Duplicates


```
import java.util.*;
public class FindDups {
        public static void main(String[] args) {
            Set<String> s = new HashSet<String>();
            for (String a : args)
                if (!s.add(a))
                    System.out.println("Duplicate detected: " + a);
            System.out.println(s.size() + " distinct words: " + s);    }
}
```

Note that the code always refers to interface type (Set):

```
Set<String> s = new HashSet<String>();
```

rather than by its implementation type (HashSet).

```
HashSet<String> s = new HashSet<String>();
```

Example: Finding Duplicates with TreeSet

Karoly.Bosa@jku.at

```
import java.util.*;
public class FindDups {
        public static void main(String[] args) {
            Set<String> s = new TreeSet<String>();
            for (String a : args)
                if (!s.add(a))
                    System.out.println("Duplicate detected: " + a);
            System.out.println(s.size() + " distinct words: " + s);    }
}
```

Note that the code always refers to interface type (Set):

```
Set<String> s = new HashSet<String>();
```

rather than by its implementation type (HashSet).

```
HashSet<String> s = new HashSet<String>();
```

Implementation HashSet makes no guarantees as to the order of the elements.
If you want the program to print the word list in alphabetical order, change the Set's implementation type from HashSet to TreeSet.

Example: Finding Duplicates Output 2

Karoly.Bosa@jku.at

Now run the program:

java FindDups i came i saw i left

The following output is produced:

Duplicate detected: i
Duplicate detected: i
4 distinct words: [came, i, left, saw]

Instead of

4 distinct words: [i, left, saw, came]

Example 2: Finding Duplicates 2

Karoly.Bosa@jku.at

```
import java.util.*;
public class FindDups2 {
        public static void main(String[] args) {
                Set<String> uniques = new HashSet<String>();
                Set<String> dups = new HashSet<String>();
                for (String a : args)
                        if (!uniques.add(a)) dups.add(a);

                // Destructive set-difference
                uniques.removeAll(dups);
                System.out.println("Unique words: " + uniques);
                System.out.println("Duplicate words: " + dups); }
 }
```

We want to know which words in the argument list occur only once and which occur more than once, but you do not want any duplicates printed out repeatedly.

Example 2: Finding Duplicates 2 Output

Karoly.Bosa@jku.at

Now run the program:

java FindDups i came i saw i left

The following output is produced:

Unique words: [left, saw, came]
Duplicate words: [i]

The List Interface

Karoly.Bosa@jku.at

- A **List** is an ordered Collection (sometimes called a *sequence,* ***resizable array***).

- Lists may contain duplicate elements.

- In addition to the operations inherited from Collection, the List interface includes operations for the following:

  - **Positional access** — manipulates elements based on their numerical position in the list
  - **Search** — searches for a specified object in the list and returns its numerical position
  - **Iteration** — extends Iterator semantics to take advantage of the list's sequential nature
  - **Range-view** — performs arbitrary *range operations* on the list.

The List Interface

Karoly.Bosa@jku.at

```
public interface List<E> extends Collection<E> {
        // Positional access
        E get(int index);
        E set(int index, E element);
        boolean add(E element);       //Add an element to the end of the List
        void add(int index, E element);
        E remove(int index);
        boolean addAll(int index, Collection<? extends E> c);

        // Search
        int indexOf(Object o);
        int lastIndexOf(Object o);

        // Iteration
        ListIterator<E> listIterator();
        ListIterator<E> listIterator(int index);
        // Range-view
        List<E> subList(int from, int to);
}
```

This slide contains only the new methods (added to the inherited ones by the List)

List Implementations

- **ArrayList** is based on conventional arrays and its is usually the better-performing implementation.
- **LinkedList** which offers better performance under certain circumstances (in case of highly variable number of elements).
- **Vector and Stack** are in the language because of historical reason and backward compatibility. They are generic data types, too.

Each implementation has a no argument (**ArrayList(), LinkedList(), Vector()**) constructor and another constructor that expects another Collection object as an argument (**conversion constructors**):

```
List<Type> l = new LinkedList<Type>(c);
```

List Itherator

Karoly.Bosa@jku.at

As before the Iterator returned by List's **iterator method**, but List also provides a richer iterator, called a **ListIterator**, which allows you to:

- traverse the list in either direction,
- modify the list during iteration, and
- obtain the current position of the iterator.

The ListIterator interface follows:

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove();
    void set(E e);
    void add(E e);
}
```

How to Use ListIterator

Traverse a list from its end:

```
…
for (ListIterator<Type> it = list.listIterator(list.size()); it.hasPrevious(); ) {
    Type t = it.previous();
    …
}
```

Examples for Range-View Operation

Karoly.Bosa@jku.at

Removing a range of elements from a List:

```
list.subList(fromIndex, toIndex).clear();
```

Similar idioms can be constructed to search for an element in a range:

```
int i = list.subList(fromIndex, toIndex).indexOf(o);
int j = list.subList(fromIndex, toIndex).lastIndexOf(o);
```

Utility Classes

Karoly.Bosa@jku.at

- java.util.Arrays
  - Filling up Arrays with values **Arrays.fill(array, value);**
  - Sorting Arrays **Arrays.sort(array);**
  - Searching in sorted Arrays **Arrays.binarySearch(array, value);**
  - Converting Arrays to Lists **Arrays.asList(array);**
- java.util.Collections
  - Sorting Lists
  - Searching in Lists
  - Searching for the maximal element in Collection
  - …

Class Collections: Singleton and Empty

Karoly.Bosa@jku.at

Class **Collections** consists exclusively of **static methods** that operate on or return collections for instance**:**

- **Special case:** Creating single a set/list/map collection **containing a single element, e.g.:**

```
Set set = Collections.singleton("Hello");
List list = Collections.singletonList("First");                    //since java 1.3
Map map = Collections.singletonMap("Key","Value"); //since java 1.3
```

- **Static constants for empty collections:**

```
Set set = Collections.EMPTY_SET;
List list = Collections.EMPTY_LIST;
Map map = Collections.EMPTY_MAP;
```

- **Other operations/methods of the class Collections see later.**

Class Collections: nCopies and fill

Karoly.Bosa@jku.at

If you need an immutable list with multiple copies of the same element:

```
List fullOfNullList = Collections.nCopies(10, null);
```

By itself, that doesn't seem to useful. However you can then make the list modifiable by passing it to another list:

```
List l = new ArrayList(fullOfNullList);
```

If you intend to Replace all of the elements of the specified list with the specified element:

```
List<String> list = new LinkedList<String>();
…
Collections.fill(list, "Hello");
```

Class Collections: Algorithms

Karoly.Bosa@jku.at

Most algorithms in the Collections class apply specifically to List:

- **sort** — sorts a List using a merge sort algorithm, which provides a fast, stable sort.
- **shuffle** — randomly permutes the elements in a List.
- **reverse** — reverses the order of the elements in a List.
- **rotate** — rotates all the elements in a List by a specified distance.
- **swap** — swaps the elements at specified positions in a List.
- **replaceAll** — replaces all occurrences of one specified value with another.
- **fill** — overwrites every element in a List with the specified value.
- **copy** — copies the source List into the destination List.
- **binarySearch** — searches for an element in an ordered List using the binary search algorithm.
- **indexOfSubList** — returns the index of the first sublist of one List that is equal to another.
- **lastIndexOfSubList** — returns the index of the last sublist of one List that is equal to another.

Class Collections: Sort


```
import java.util.*;
public class Sort {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
         Collections.sort(list);
        System.out.println(list);
    }
}
```

Let's run the program.

java Sort i walk the line

The following output is produced.

[i, line, the, walk]

Of course, this sort works only if the elements of the list implement the **Comparable interface** as in case of our generic *Quick Sort program.*