Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

Programming Techniques in Java

Collections

I. Salomie, C.Pop
2017

Introduction

- Collection (or container)
  - Object that contains other objects
    - collection elements
  - Collection elements can be added / removed / manipulated in the collection
  - In Java Collection Framework collection elements are objects (not primitive types)
- Main classification criteria
  - Ordered / un-ordered
  - Concurrent / non-concurrent
  - Duplicated elements are allowed or not

JCF

- JCF - a unified architecture for representing and manipulation collections
- JCF
  - Interfaces
    - ADT descriptions supported by Java Collection Framework
    - Allow collections to be manipulated independently of their details of implementation
  - Implementations
    - Concrete implementations of the collection interfaces
  - Algorithms
    - Perform useful computations on collections

JCF benefits

- Reduces programming effort
  - Allows the programmer to concentrate on the important parts of the programs
- Increases program speed and quality
  - Provides high-performance, high-quality implementations of useful data structures and algorithms
  - Interchangeable implementations - switching between collections when needed
  - More time to improving programs' quality and performance
- Allows interoperability among unrelated APIs
  - Collection interfaces - interoperability main support
  - No adapters necessary
- Fosters software reuse
  - 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

Collection types

- Bag (also called Multiset)
  - Most general form of collections
    - Unordered collection
    - Duplicate elements
  - Bag behavior
    - Java Collection interface
- Sets
  - No duplicate elements
  - Unsorted sets
    - Unordered collection
  - Sorted sets (or ordered sets)
  - Set behavior
    - Java Set type interfaces
- Lists (or sequences)
  - Ordered collection of elements
  - Indexed elements (starting from 0)
  - Duplicated elements - allowed
  - List behavior
    - Java List interface
- Maps
  - Also known
    - Functions, Dictionaries , Associative arrays
  - Unordered collection of association (key, value)
    - Keys must be unique
    - Value - any entity
  - Sorted maps (ordered maps)
  - Map behavior
    - Java Map type interfaces
  - Map example
- Note. Array is directly implemented by JDK

Implementation DS support

Arrays

- Directly implemented in hardware
- Property of random-access memory
  - Fast for accessing elements by position and for iterating
  - Slower at inserting / removing elements at arbitrary positions (due to shifting)
- Backing structures for
  - ArrayList, many Queue / Deque and Hashtables implementations

Linked Lists

- Chains of linked cells
- Access by position is slow
- Insertion / Removal perform in constant time
- Backing structures for
  - LinkedList, LinkedBlockingQueue, ConcurrentLinkedQueue
  - HashSet and LinkedHashSet

Implementation DS support

Hashtables

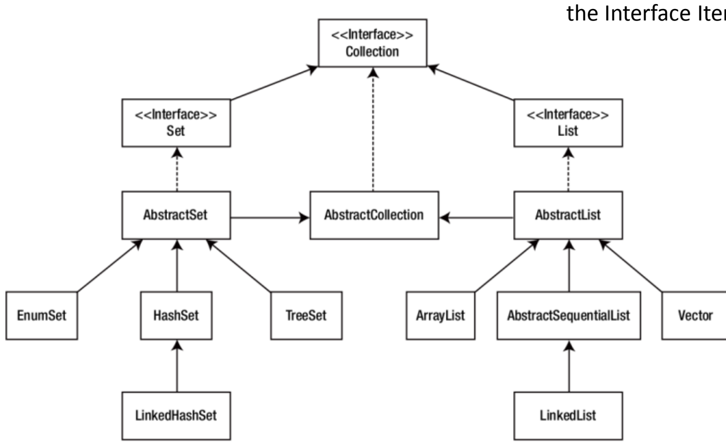
- Stores elements indexed on their content rather than on integer-valued index as with lists
- Access
  - No support for accessing elements by position
  - Access by content is usually fast
  - Insertion and removal are fast as well
- Backing structures for
  - Many Set and Map implementations
    - HashSet, LinkedHashSet
    - HashMap, LinkedHashMap, WeakHashMap, IdentityHashMap, ConcurrentHashMap

Trees

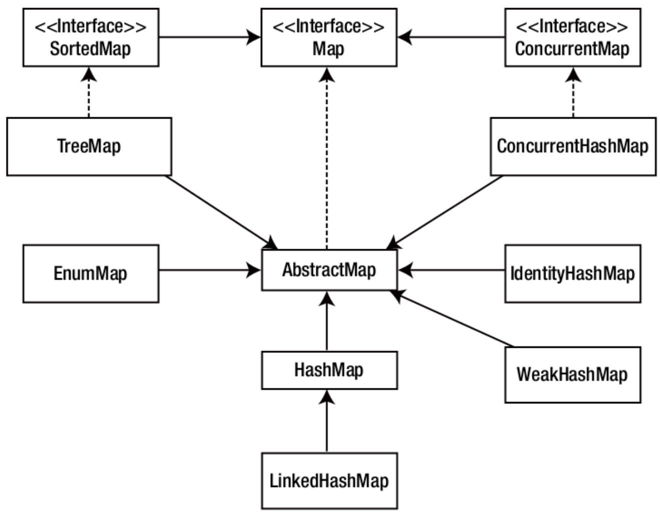
- Content based organized structure
  - Can store and retrieve elements in sorted order
  - Relatively fast for operations of insert / remove
  - Access elements by content
- Backing structures for
  - TreeSet, TreeMap
  - PriorityQueue, PriorityBlockingQueue

Collection and Map Hierarchies

Interface Collection implements the Interface Iterable<E>



Collection and Map Hierarchies



Evolution

- Earliest Java versions: Vector, Hashtable, Stack
  - All synchronized
  - Performance penalty
- Java 2 (1998)
  - Most of the current collections were defined
  - New added classes - No thread safe (serial access to collections being necessary)
- Java 5 (2004)
  - Major changes, some new syntax
  - Prior Java 5 code regarding collections generate errors
  - Autoboxing wraps primitives and allows them to be directly added to collections
  - Generics was introduced in Java 5; Collections can be more appropriate defined (before collections were collecting Object and cast was necessary when retrieving the items)
- Java 7 (2011)
  - Diamond operator (generic type can be specified only in the left side of collection instantiation)
- Java 8 (2014)
  - Collection processing using Lambda expressions, pipes and streams

Class for examples

```
public class Student {
  private int studentID;
  private String firstName;
  private String lastName;
  public Student(int id, String fname, String lname) {
    studentID = id;
    firstName = fname;
    lastName = lname;
  }
  public int getStudentID() { return studentID; }
  public String getFirstName() { return firstName; }
  public String getLastName() { return lastName; }
  public String getFullName() { return getFirstName() + " " + getLastName(); }
}
```


The arrays

- Directly implemented in JDK
- Converting Collection to Array

```
List<String> lst = new LinkedList<>();
// populate the List
```

- Technique 1

```
String[]  a1 = (String[]) lst.toArray();
```

- Technique 2

```
Set<String> set = …
String[]  a2 = set.toArray(new String[set.size()]);
```

- Array as Collection

```
String[] arr = … ;
List<String> lst = Arrays.asList(arr);
```

The methods:

```
Object[] toArray()
<T> T[] toArray(T[] a)
```

Collection
Methods

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

// adding elements

public boolean add(E e)

public boolean addAll(
  Collection<? extends E> c)

// removing elements

public boolean remove(Object o)

public void clear()

public boolean removeAll(Collection<?> c)

public boolean retainAll(Collection<?> c)

(cont.)

// querying collection contents

public boolean contains (Object o)

public boolean containsAll(Collection<?> c)

public boolean isEmpty()

public int size()

// comparison operations

boolean equals(Object o)

int hashCode()

// transform / processing

public Iterator<E> iterator()

public Object[] toArray()

public <T> T[] toArray(T[] t)

}

Collection Constructors

Default constructors
public InterfaceType<E> = new ConcreteClass<>()

Example
public Set<String> = new HashSet<>();

Conversion constructors
public InterfaceType<E> = new ConcreteClass(Collection<? extends E> c)

- All JCF Collections have a constructor with arguments taking Collection as argument
  - This initializes the new collection as to contain all of the elements in the argument collection
  - Allows the conversion of collection types

Example
- Suppose you have a **List lst** already populated with Strings

Set<String> ss = new SortedSet(lst);
- lst will be automatically converted to a SortedSet object;
- All duplicated elements will be eliminated

Collection

- Remember
  - Whenever working with Collections and override **equals** you should also override **hashCode**

Class Collections

- Defined in java.util
- Contains
  - Static methods
    - operate on collections or
    - return collections
  - Polymorphic algorithms that operate on collections
  - "Wrappers", which return a new collection
  - Few other odds and ends.
- Examples of static methods
  - binarySearch, copy, disjoint, max, min, replaceAll, reverse, sort, shuffle
- The methods of this class all throw a NullPointerException
  - when collections or class objects provided to them are null

Iterating Collections
Ways of traversing collections

External Iterators
- Regular loops for index associated collections (such as List)
- Iterator
- for-each loop
- forEach() method

Internal Iterators
- Defined by using Streams

Iterating Collections
Interfaces Iterable and Iterator

```
public interface Iterable<E> {
  Iterator<E> iterator();
  default void forEach(Consumer<? super E> action);
  default Spliterator<E> spliterator();
}
```

iterator()

Returns an iterator over elements of type E

forEach()

Performs **action** for each element of the Iterable until all elements have been processed or the action throws an exception

The default implementation:

for (E e : this) action.accept(t);

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

public interface Iterator<E> {
  boolean hasNext();
  E next();
  default void remove(); // optional
  default void forEachRemaining(
          Consumer<? super E> action);
}
```

- **remove()** – removes the last returned element
  - Should be called once per next (exception IlelgalState is thrown)
- **forEachRemaining()** – introduced by Java 8
  - Acts upon each Collection element that was not yet processed (applicator)

Iterating Collections
Iterate using Iterator

Main operations
- Check if there are elements not been yet accessed using this iterator
- Get the next element
- Remove the last accessed element

Example (hasNext, next)

```
List<String> names = … // populate the list;
Iterator<String> it = names.iterator();
  while(it.hasNext()) {
    // Get the next element from the list
    String name = it.next();
    // … process the name somehow
    System.out.println(name);
  }
```

Example (remove)

```
List<String> names = … // populate the list;
Iterator<String> it  = names.iterator();

// Iterate
while(it.hasNext()) {
  String name = it.next();
  // Remove if less than two chars
  if (name.length() <= 2) {
    it.remove();
  }
}
```


Iterating Collections
Iterate using Iterator

Main operations
- Check if there are elements not been yet accessed using this iterator
- Get the next element
- Remove the last accessed element

Example (hasNext, next)

```
List<String> names = … // populate the list;
Iterator<String> it = names.iterator();
  while(it.hasNext()) {
    // Get the next element from the list
    String name = it.next();
    // … process the name somehow
    System.out.println(name);
  }
```

Example (remove)

```
List<String> names = … // populate the list;
Iterator<String> it  = names.iterator();

// Iterate
while(it.hasNext()) {
  String name = it.next();
  // Remove if less than two chars
  if (name.length() <= 2) {
    it.remove();
  }
}
```

Iterating Collections
Iterate using Iterator

```
List<String> names = // … populate the list;
// Get an iterator for the list
Iterator<String> it = names.iterator();
// Print the names in the list
it.forEachRemaining(System.out::println);
```

- Method reference System.out::println acts as Consumer
- Reduced code size by eliminating a loop with hasNext() and next()

- How to use it (on short)

```
List<String> names = // … populate the list;
// Print all elements of the names list
names.iterator().forEachRemaining(System.out::println);
```

fast-fail concurrent iterators

- Many iterators running concurrently over the same collection
- If the collection is modified other way than using **remove()** method (of the same iterator) => throw **ConcurrentModificationException** when accessing the next element
- **Note**
  - An iterator is a one-time object
  - An iterator cannot be reset
  - New iterator needs to be obtained to reiterate over the same collection

Iterating Collections
Iterate using for-each loop and forEach method

for-each loop

- Hides the logic of creating an Iterator and executing hasNext and next operations;
- Can be used to iterate over collection type class that implement interface **Collection**
- It is actually implemented by instantiating an iterator and calling hasNect() and next()

```
Collection<T> col = // ... get a collection;
for(T element : col) {
  // execute the block for each col element
  // element refers the current col item
}
```

Limitations

- Collection must be iterated from the start to the end
- Cannot be used to remove elements

```
List<String> names = get a list;
for(String name : names) {
  // Throws a ConcurrentModificationException
  names.remove(name);
}
```

forEach method (Java 8)

- forEach(Consumer<? super T> action)
- Applicator - available to all Collection types
- Defined by interface Iterable
- Similar to forEachRemaining but applies **action** upon all collection elements
- How to use it

```
List<String> names = // … populate the list;
// Print all elements of the names list
names.forEach(System.out::println);
```

ConcurrentModificationException

- fast-fail concurrent iterators
- Concurrent execution iterators over a Collection
- Exception is thrown if the collection is modified by any identifier (except using remove)
- Iterator – one-time object (cannot be reset)
  - A new Iterator must be created

Consumer and Supplier
Review

- Functional Interfaces defined by Java 8 (java.util.function)
- Consumer - a function that takes one argument of an arbitrary type and produce no result
- Used with Streams of data to execute a given action for every stream element
- List<Student> ls = …

```
ls.stream().forEach((u) -> System.out.println("Name: " + u.getName()));
```

- Supplier - a function that takes no arguments and produce a value of an arbitrary type
- The value can be retrieved using get() on the Supplier

Example 1

```
Supplier<Student> studSupplier = Student::new;
Student std = studSupplier.get();
```

Example 2

```
Supplier<Student> studSupplier = this::generateStudent;
Student std = studSupplier.get();
private Student generateStudent() { return new Student(); }
```


Set

- Interface Set (inherits interface Collection)
- Modes mathematical set
- Unique elements
- No effect when adding an already existing element
- **add** and **addAll** are specific implemented
- No guarantee on element order in set
  - Add element in one order and retrieve in other order
  - However, order sets can also be defined
  - Ordered Set behavior is described by the interface SortedSet (inherits interface Set)

Set

Main mathematical operations on sets union, intersection, difference

```
// Suppose s1 and s2 are sets;
// Results will be stored in s1

// Union
s1.add(s2);

// Intersection
s1.retainAll(s2);

// Difference
s1.removeAll(s2);

// Keep unmodified original sources – make a copy before operation
```

Example

```
public static void performUnion(Set<String> s1,
                                Set<String> s2) {
  Set<String> s1Unions2 =
            new HashSet<>(s1); // copy of s1
  s1Unions2.addAll(s2);
  System.out.println("s1 union s2: " + s1Unions2);
}
```

Note. When passing an immutable set (or any collection) to a method you may use (for example in the case of Set s1):

Collections.unmodifiableSet(s1)

- Subset testing

```
// is s1 subset of s2?
If (s2.containsAll(s1) …
```


Set

Main Set implementations

- HashSet (no order regarding add / retrieve)
- LinkedHashSet (order regarding add / retrieve)
- Sorted Sets

Set from Collection

```
Collection<T> c = // … create and populate a collection with duplicates
Set<T> noDup = new HashSet<T>(c);
Set<T> noDupOrder = new LinkedHashSet<T>(c);
```

Idiom encapsulation

```
public static <E> Set<E> removeDups(Collection<E> c) {
   return new LinkedHashSet<E>(c);
}
```

Set
HashSet

- Most common set implementation
- No guarantee concerning the order of iteration
- Unsynchronized and not thread safe
- Iterators are of type fail-first
- Support data structure: hashtable
  - Array where the elements are stored at a position derived from content
  - If no collisions - the cost of inserting / retrieving elements is constant
  - The cost increases as more collisions are happening (i.e. as more items are added to the set, linked lists being used to solve the collisions)
  - This can be improved by **rehashing** - copy the hashtable to a larger one when the **load factor** is overtaken

Set
HashSet

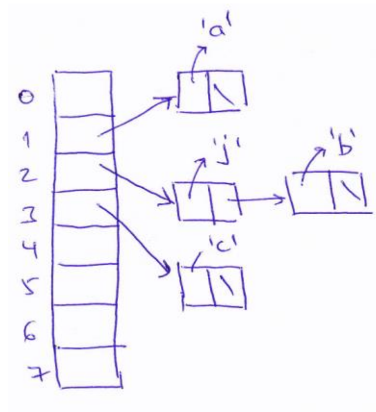
- An element's position in a hash table is calculated by a hash function of its contents
- Hash functions – good spread of results (hash codes) over the stored element values
- Example of hashCode for a String class

```
int hash = 0;
for (char ch : str.toCharArray()) {  hash = hash * 31 + ch; }
```

- Traditionally, hash tables obtain an index from the hash code by taking the remainder after division by the table length
- JCF classes use bit masking rather than division
- Collisions happened => colliding elements should be kept at the same table location (bucket) = > linked lists of colliding values

Set
HashSet

```
Set<Character> s1 = new HashSet<Character>(8);
s1.add('a');
s1.add('b');
s1.add('j');
s1.add('c');
```



- The index values of the table elements was calculated by using the bottom three bits (for a table of length 8) of the hash code of each element.
- In this implementation, a Character's hash code is just the Unicode value of the character it contains
- **Note**. HashSet implementation uses a private HashMap, so each cell in the chain actually contains a key and a value. The diagram shows only the key (in the case of Set all values for a key should be the same)

Set
HashSet

Constructors

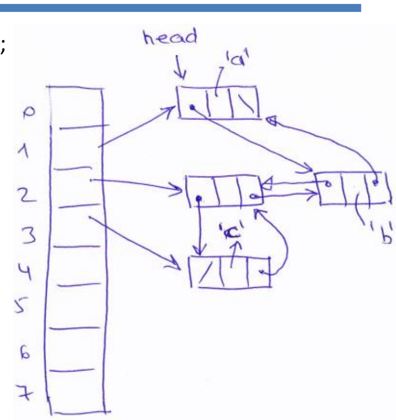
```
// default and conversion constructors
public HashSet()
public HashSet(Collection<? extends E> c)

// specific constructors (both create empty sets)
public HashSet(int initialCapacity)  // next largest (power of 2 -1)
public Hashet(int initialCapacity, float loadFactor)
```

Sets
LinkedHashSet

- Inherits from HashSet
- Guarantees that the iterators will return the set elements in the order they were first added
  - it maintains a linked list
  - Iterator operation **next** always perform in constant time (due to linked list)

Set
HashSet

Set<Character> s2 = new LinkedHashSet<Character>(8);
Collections.addAll(s2, 'a', 'b', 'j', 'c');
// iterators of a LinkedHashSet return
// their elements in proper order:
assert s2.toString().equals("[a, b, j,c]");



Sets
Sorted Set

- Behavior is specified by the interface **SortedSet**
- Imposes order on its elements
- The order can be: natural order or imposed by a Comparator object
- Natural Order
  - SortedSet elements implement the Comparable interface
  - The order is determined by the method compareTo
- Custom order
  - The class that implement SortedSet provides the Comparator object as constructor parameter
  - If Comparator is specified, it takes over the Comparable (even if the class implements the Comparable interface)

- Note. If class implementing the SortedSet is not implementing Comparable, nor defining a Comparator object => no item can be added to the collection => generates ClassCastException

Sets
Sorted Set

```
public interface SortedSet<E> extends Set<E> {
   // Range-view
   // inclusive lower bound, exclusive higher bound
   SortedSet<E> subSet(E fromElement, E toElement);
   SortedSet<E> headSet(E toElement);
   SortedSet<E> tailSet(E fromElement);

   // Endpoints
   E first();
   E last();

   // Comparator access
   Comparator<? super E> comparator();
}
```

Sets
TreeSet a Sorted Set implementation

- TreeSet constructors
  - Default constructor
  - Copy constructor

  // builds a new empty set which will be sorted using the supplied comparator
  - TreeSet(Comparator<? super E> c)

- String is a class that implements interface Comparable

SortedSet<String> sortedNames = new TreeSet<>();
// … add names using sortedNames.add(…);
// Print the sorted set of names
System.out.println(sortedNames);

Sets
TreeSet a Sorted Set implementation

Class Student enhanced with equals, hashCode and toString
Student is not implementing Comparable interface

```
public class Student {
  private int studentID;
  private String firstName;
  private String lastName;
  public Student(int id, String fname, String lname) {
    studentID = id;
    firstName = fname;
    lastName = lname;
  }
  public int getStudentID() { return studentID; }
  public String getFirstName() { return firstName; }
  public String getLastName() { return lastName; }
  public String getFullName() { return getFirstName()
+ " " + getLastName(); }
```

```
@Override
public boolean equals(Object o) {
  if (!(o instanceof Student)) { return false; }
  // id must be the same for two Students to be equal
  Student s = (Student) o;
  if (this.id == s.getStudentID()) { return true; }
  return false;
}
@Override
public int hashCode() {
  // A trivial implementaiton
  return this.studentID;
}
@Override
public String toString() {
   return "(" + studentID + ", " + getFullName() + ")";
}
```

Sets
TreeSet a Sorted Set implementation

```
Set<Student> students = new TreeSet<>();
students.add(new Student(…));
// exception is thrown
```

- Below is used a method reference to generate a lambda expression for creating the Comparator object

```
SortedSet<Student> studentsSortedByName =
        new TreeSet<>(Comparator.comparing(Students::getFullName));
```

- Another Comparator object can be created to compare students by ID

```
SortedSet<Student> studentsSortedByID =
        new  TreeSet<>(Comparator.comparing(Students::getStudentID));
```

- Sorting a set of strings based on their length

```
SortedSet<String> names =
        new TreeSet<>(Comparator.comparing(String::length));
```

- **Note**.
- **comparing** is a static method of the *Functional Interface* **Comparator**
- Method **comparing** accepts a function that extracts a Comparable sort key from a type T and returns a Comparator<T> that will compare by that sort key

Sets
TreeSet a Sorted Set implementation

Working with subsets – method subset of interface SortedSet

```
// Create a sorted set of names and populate it with names
SortedSet<String> names = new TreeSet<>();
// … names.add("Vasile");
// Print the sorted set
System.out.println("Sorted Set: " + names);

// Print the first and last elements in the sorted set
System.out.println("First: " + names.first());
System.out.println("Last: " + names.last());

// subsets
SortedSet ss1= names.headSet(name1);
System.out.println("Head Set Before name1: " + ss1);
SortedSet ss2 = names.subSet(name1, name2);
System.out.println("Subset between name1 and name2 (exclusive): " + ss2);
// Note the trick name2 + "\0" to include name2 in the subset
SortedSet ss3 = names.subSet(name1, name2 + "\0");
System.out.println("Subset between name1 and name2(Inclusive): " + ss3);;
SortedSet ss4 = names.tailSet(name2);
System.out.println("Subset from name2 onwards: " + ss4);
```

Sets
NavigableSet extends Sorted Set

- Extends SortedSet functionality (since Java 6)
- Improves range view methods of SortedSet

// Getting range views
NavigableSet<E> **subSet**(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)

NavigableSet<E> **headSet**(E toElement, boolean inclusive)

NavigableSet<E> **tailSet**(E fromElement, boolean inclusive)

// Getting closest matches
E **ceiling**(E e)
 // return the least element in this set greater than
 // or equal to e, or null if there is no such element
E **floor**(E e)
 // return the greatest element in this set less than
 // or equal to e, or null if there is no such element
E **higher**(E e)
 // return the least element in this set strictly
 // greater than e, or null if there is no such element
E **lower**(E e)
 // return the greatest element in this set strictly
 // less than e, or null if there is no such element

Usefulness for short distance navigation

Example

Find, in a sorted set of strings, the last three strings in the subset that is bounded above by "x-ray", including that string itself if it is present in the set.

```
NavigableSet<String> stringSet = new TreeSet<String>();
Collections.addAll(stringSet, "abc", "cde", "x-ray" ,"zed");
String last = stringSet.floor("x-ray");
assert last.equals("x-ray");
String secondToLast = last == null ? null :
        stringSet.lower(last);
String thirdToLast = secondToLast == null ? null :
        stringSet.lower(secondToLast);
assert thirdToLast.equals("abc");
```


Sets
NavigableSet extends Sorted Set

- Navigate the set in reverse order

NavigableSet<E> **descendingSet**()
 // return a reverse-order view of
 // the elements in this set
Iterator<E> **descendingIterator**()
 // return a reverse-order iterator

- Example

```
NavigableSet<String> headSet =
          stringSet.headSet(last, true);
NavigableSet<String> reverseHeadSet =
          headSet.descendingSet();
assert reverseHeadSet.toString().equals("[x-ray, cde, abc]");
String conc = " ";
for (String s : reverseHeadSet) {
  conc += s + " ";
}
assert conc.equals(" x-ray cde abc ");
```

- If set structural changes is required we may use explicit iterator

```
for (Iterator<String> itr =
  headSet.descendingIterator(); itr.hasNext(); ) {
      itr.next();
      itr.remove();
}
assert headSet.isempty();
```

List

- List
  - Ordered (topological) collection
  - Also called sequence
  - May contain duplicate elements
- Inherits Collection defined operations
- Additional operations for
  - Positional access
    - Manipulates elements based on their position in the list
  - Search
    - Search for specified object and returns position
  - Iteration
    - Extends Iterator semantics to take advantage of List's sequential nature
  - Range-view
    - Performs arbitrary range operations on the list

```
public interface List<E> extends Collection<E> {
  // Positional access
  E get(int index);
  E set(int index, E element);        //optional
  boolean add(E element);             //optional
  void add(int index, E element);  //optional
  E remove(int index);                //optional
  boolean addAll(int index,
     Collection<? extends E> c);  //optional
  // Search
  int indexOf(Object o);
  int lastIndexOf(Object o);
  // Iteration
  ListIterator<E> listIterator();
  ListIterator<E> listIterator(int index);
  // Range-view [from, to)
  List<E> subList(int from, int to);
}
```


List
List implementations

- *ArrayList*
- *LinkedList*

List
Collection operations (meaning for List)

***addAll* operation**

- Always append to the end of the list
- List concatenation

  list1.addAll(list2); // list concatenation

- List concatenation - nondestructive approach (uses the ArrayList's standard conversion constructor)

```
List<Type> list3 =
        new ArrayList<Type>(list1);
list3.addAll(list2);
```

***remove* operation**

- Always removes the first occurrence of the specified element
- In ArrayList implementation (consider lst as a populated ArrayList)

lst.remove(0)

- Removes and shifts left all array elements

Wrong approach

- Suppose int[] deleteIndices contains a set of indices calculated earlier and you would like to remove the corresponding objects from an ArrayList

```
int[] deleteIndices;
List myList;
// Populate list, get indices of objects to be deleted
...
for (int i = 0; i < deleteIndices.length; i++) {
myList.remove(deleteIndices[i]);
}
```

Correct approach

```
for (int i = deleteIndices.length - 1; i >= 0; i--) {
myList.remove(deletedIndices[i]);
}
```


List
Iterators

- Inherited *iterator* operation
  - Return the list elements in proper sequence
- *ListIterator* - List enhanced iterators
  - Allows:
    - List traversing in both directions
    - Modify list during iteration
    - Get current iteration position

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

List
Iterators

Cursor
- Always between two list elements
  Elem1 | Elem2
  cursor
- Elem 1 - returned after calling previous
- Elem2 - returned after calling next
- A call to *nextIndex()* returns the index of the element that would be returned by a subsequent call to *next()*
- A call to *previousIndex()* returns the index of the element that would be returned by a subsequent call to *previous()*
- There are n+1 index values
  - They correspond to n+1 gaps between elements
  - Starting gap is before the first element
  - Last gap is after the last element

Backwards iteration idiom

```
for (ListIterator<T> it = lst.ListIterator(lst.size());
  it.hasPrevious(); ) {
   T t = it.previous();
  ...
}
```

- *nextIndex* always returns the number returned by *previousIndex* plus 1
  - Boundary cases:
  - Cursor is before the first element
    - *previousIndex* returns -1
  - Cursor is after the last element
    - *nextIndex* returns *list.size()*

List
Iterators

- Intermixing calls is allowed in a careful way
  - The first call to *previous* returns the same element as the last call to *next*
  - The first call to *next* returns after a sequence of calls to *previous* returns the same element as the last call to *previous*
- How to use these knowledge?
  - Report the position where something was found
  - Record the position of the ListIterator so that another ListIterator can be created having a similar cursor position
- Operation *remove*
  - Removes the last element returned by *next* or *previous*
- ListIterator interface also provides the operations *set* and *add*
- *set*
  - Overrides the last element returned by *next* or *previous* (see example)
- *add*
  - Inserts a new element into the list immediately before the current cursor position (see example)

List
Iterators

Polymorphic algorithm to replace all occurrences of the one parameter with the other one

- Using *set method*

```
public static <E> void replace(List<E> list, E val, E newVal) {
   for (ListIterator<E> it = list.listIterator(); it.hasNext(); )
      if (val == null ? it.next() == null : val.equals(it.next()))
         it.set(newVal);
}
```

- Using *add method*

```
public static <E> void replace(List<E> list, E val, List<? extends E> newVals) {
   for (ListIterator<E> it = list.listIterator(); it.hasNext(); ) {
      if (val == null ? it.next() == null : val.equals(it.next())) {
         it.remove();
         for (E e : newVals) it.add(e);
      }
   }
}
```


List
Range-view operations

subList(int fromIndex, int toIndex)

- Returns a list view from fromIndex (inclusive) to toIndex (exclusive)
- Any operation that expects a List can be used as a range operation by passing a subList view (instead of a whole List)
- Care should be taken when operating with subList
- Examples

```
// removing a range of elements from a list
list.subList(fromIndex, toIndex).clear();

// determine the index of a certain element in the
   subList
int i = list.subList(fromIndex, toIndex).indexOf(o);
int j = list.subList(fromIndex,
   toIndex).lastIndexOf(o);
```

Example

- Polymorphic algorithm for generating a hand from a deck of cards

```
public static <E> List<E> dealHand(List<E> deck, int n) {
   int deckSize = deck.size();
   List<E> handView = deck.subList(deckSize - n,
         deckSize);
   List<E> hand = new ArrayList<E>(handView);
   handView.clear();
   return hand;
}
```

Notes

- Removes the hand from the end of the deck
- For common List implementations, removing from the end is faster than removing from front

List specific algorithms of class Collections

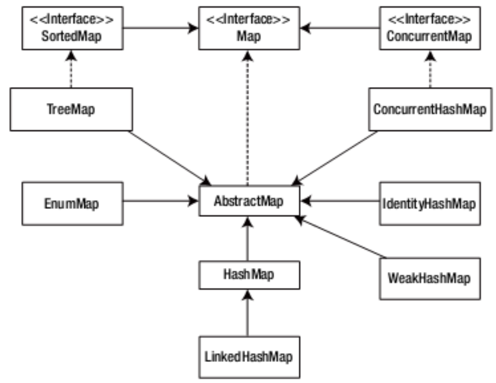
* sort - sorts a List using a merge sort algorithm, which provides a fast, stable sort. (A stable sort is one that does not reorder equal elements.)
* shuffle - randomly permutes the elements in a List.
* reverse - reverses the order of the elements in a List.
* rotate - rotates all the elements in a 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.

Map

- Map
  - Object that maps keys to values
  - A (key, value) pair is an entry in the Map
  - No duplicate keys
  - One key maps to at most one value
- A Map is a collection of Entries
- An Entry is specified by the interface Map.Entry
  - Map.Entry - inner interface of the interface Map
- No iterator method
  - keySet, entrySet methods return Set
  - values method return Collection
  - Set and Collection could be iterated

Map

```
public interface Map<K,V> {
   // Basic operations
   V put(K key, V value);
   V get(Object key);
   V remove(Object key);
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   int size();
   boolean isEmpty();
   // Bulk operations
   void putAll(Map<? extends K, ? extends V> m);
   void clear();
   // provides Collection Views
   public Set<K> keySet();
   public Collection<V> values();
   public Set<Map.Entry<K,V>> entrySet();
   // Interface for entrySet elements
   public interface Entry {
      K getKey();
      V getValue();
      V setValue(V value);
   }
}
```



Main Map Implementations

Unsorted
- HashMap
- LinkedHashMap (inherits from HashMap)

Sorted
- TreeMap – ordered by key

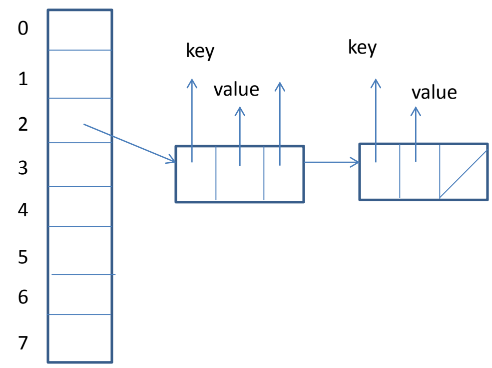
- Every implementtaions has at least two constructors
- Example for HashMap

```
// creates an empty map
public HashMap()

// build a HashMap from an input map
// equivalent to building an amply map + putAll
public HashMap(Map<? extends K,? extends V> m)
```

Main Map Implementations
HashMap

- Additional constructors

public HashMap(int initialCapacity)
public HashMap(int initialCapacity, float loadFactor)



Main Map Implementations
LinkedHashMap

- Refines the contract of HashMap
- Guarantees the order in which iterators return its elements
- Unlike LinkedHashSet LinkedHashMap offers a choice of iteration orders
  - in the order in which they were inserted in the map
  - in the order in which they were accessed (from least-recently to most-recently accessed)

public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)

  - accessOrder = false => insertion-order map

Main Map Implementations
SortedMap - TreeMap

- Interface SortedMap inherits from the interface Map
- Stores entries in order by key
- Natural Order – defined by Comparable interface on Keys; If the Keys don't implement the Comparable interface a Comparator object should be used
- TreeMap implements the interface SortedMap
-

Main Map Implementations
SortedMap - TreeMap

SortedMap methods

- K firstKey()
- K lastKey()
- SortedMap<K, V> headMap(K toKey)
  - Returns a view of the SortedMap
  - Entries have keys less than the specified toKey.
  - If adding a new entry to the view, its key must be less than the specified toKey (otherwise exception)
  - The view is backed by the original SortedMap
- SortedMap<K, V> tailMap(K fromKey)
  - Similar to headMap
- SortedMap<K, V> subMap(K fromKey, K toKey)
  - Returns a view of the SortedMap
  - Entries have keys ranging from fromKey (inclusive) and toKey (exclusive).
  - Original SortedMap backs the partial view of the SortedMap.
  - Any changes made to either map will be reflected in both

Views on Collection supported by class Collections

- Read only views (unmodifiable view) on Collections

<T> Collection<T> unmodifiableCollection(Collection<? extends T> c)

<T> List<T> unmodifiableList(List<? extends T> list)

<K,V> Map<K,V> unmodifiableMap(Map<? extends K,? extends V> m)

<T> Set<T> unmodifiableSet(Set<? extends T> s)

<K,V> SortedMap<K,V> unmodifiableSortedMap(SortedMap<K,? extends V> m)

- Synchronized View (thread safe) of a Collection

<T> Collection<T> synchronizedCollection(Collection<T> c)

<T> List<T> synchronizedList(List<T> list)

<K,V> Map<K,V> synchronizedMap(Map<K,V> m)

<T> Set<T> synchronizedSet(Set<T> s)

<T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)

<K,V> SortedMap<K,V> synchronizedSortedMap (SortedMap<K,V> m)

Views on Collection supported by class Collections

- Creating empty collections

<T> List<T> emptyList()

<K,V> Map<K,V> emptyMap()

<T> Set<T> emptySet()

<T> Iterator<T> emptyIterator()

<T> ListIterator<T> emptyListIterator()

- Use
- Suppose you have a method m1(Map<String,String> map)
- Call it with an empty map
- m1(Collections.emptyMap());