

Java Collection Framework

Version 2 - April 2014






© Maurizio Morisio, Marco Torchiano, 2014



Attribution–NonCommercial–NoDerivs 2.5

- You are free: to copy, distribute, display, and perform the work

Under the following conditions:

-  **Attribution.** You must attribute the work in the manner specified by the author or licensor.
-  **Noncommercial.** You may not use this work for commercial purposes.
-  **No Derivative Works.** You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license) found at the end of this document

Framework

- Interfaces (ADT, Abstract Data Types)
- Implementations (of ADT)
- Algorithms (sort)

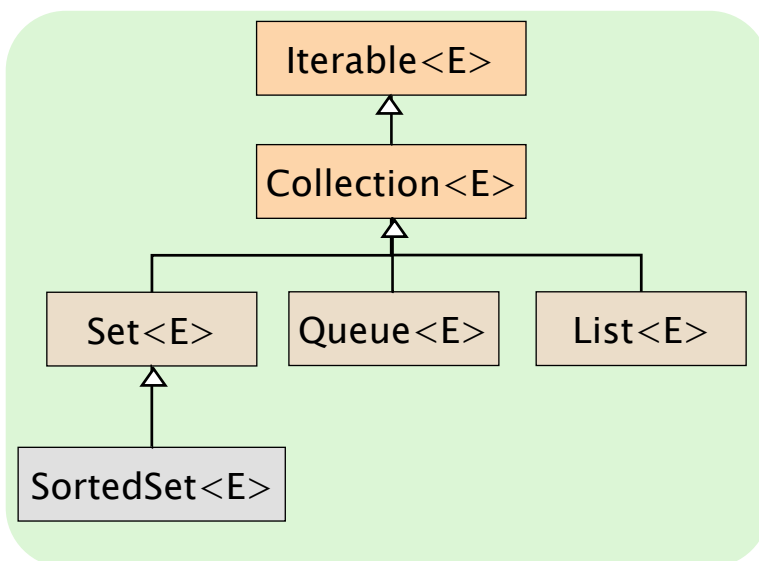
- `java.util.*`

- After Java 5 release
 - ◆ Lots of changes about collections

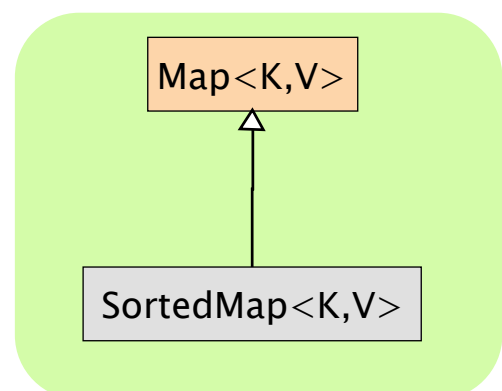
SoftEng
<http://softeng.polito.it>

3

Interfaces



Group containers

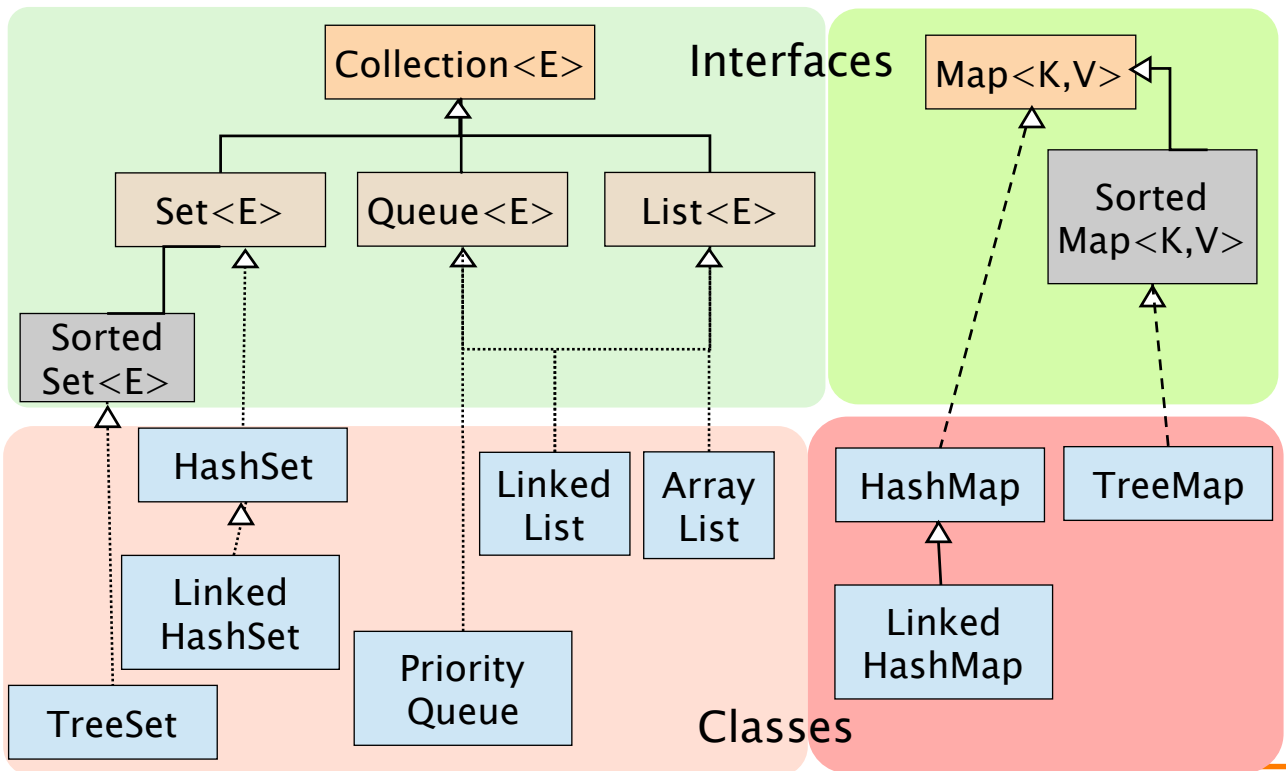


Associative containers

SoftEng
<http://softeng.polito.it>

4

Implementations



5

Internals

data structure

	Hash table	Resizable array	Balanced tree	Linked list	Hash table Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

interface

classes

Iterable

- Container of elements that can be iterated upon
- Contains a single method:
`Iterator<E> iterator()`
 - ◆ It returns the iterator on the elements

Iterators and iteration

- A common operation with collections is to iterate over their elements
- Interface Iterator provides a transparent means to cycle through all elements of a Collection
- **Keeps track of last visited** element of the related collection
- Each time the current element is queried, it **moves on automatically**

Iterator

- Is the class that allows the iteration on the elements of a collection
- Two main methods:
 - ◆ **boolean hasNext()**
 - Checks if there is a next element to iterate on
 - ◆ **E next()**
 - Returns the next element and advances by one position
 - ◆ **void remove()**
 - Optional method, removes the current element

Iterator examples

Print all objects in a list

```
Iterable<Person> persons =  
    new LinkedList<Person>();  
...  
for(Iterator<Person> i = persons.iterator();  
    i.hasNext(); ) {  
    Person p = i.next();  
    ...  
    System.out.println(p);  
}
```

Iterator examples

The for-each syntax avoids
using iterator directly

```
Iterable<Person> persons =
    new LinkedList<Person>();
...
for(Person p: persons) {
    ...
    System.out.println(p);
}
```

Iterator examples (until Java 1.4)

Print all objects in a list

```
Collection persons = new LinkedList();
...
for(Iterator i= persons.iterator(); i.hasNext(); ) {
    Person p = (Person)i.next();
    ...
}
```

Collection

- **Group** of elements (**references** to objects)
- It is not specified whether they are
 - ◆ Ordered / not ordered
 - ◆ Duplicated / not duplicated
- Following constructors are common to all classes implementing Collection
 - ◆ `C()`
 - ◆ `C(Collection c)`

Collection interface

- `int size()`
- `boolean isEmpty()`
- `boolean contains(E element)`
- `boolean containsAll(Collection<?> c)`
- `boolean add(E element)`
- `boolean addAll(Collection<? extends E> c)`
- `boolean remove(E element)`
- `boolean removeAll(Collection<?> c)`
- `void clear()`
- `Object[] toArray()`
- `Iterator<E> iterator()`

Collection example

```
Collection<Person> persons =
    new LinkedList<Person> ();
persons.add( new Person("Alice") );
System.out.println( persons.size() );
Collection<Person> copy =
    new TreeSet<Person> ();
copy.addAll( persons ); //new TreeSet( persons )
Person[] array = copy.toArray();
System.out.println( array[0] );
```

Map

- An object that associates **keys to values** (e.g., SSN \Rightarrow Person)
- Keys and values must be **objects**
- **Keys** must be **unique**
- Only one value per key

- Following constructors are common to all collection implementers
 - ♦ **M()**
 - ♦ **M(Map m)**

Map interface

- V **put**(K key, V value)
- V **get**(K key)
- Object **remove**(K key)
- boolean **containsKey**(K key)
- boolean **containsValue**(V value)
- public Set<K> **keySet**()
- public Collection<V> **values**()
- int **size**()
- boolean **isEmpty**()
- void **clear**()

Map example

```
Map<String, Person> people =
    new HashMap<String, Person> ();
people.put( "ALCSMT", //ssn
    new Person("Alice", "Smith") );
people.put( "RBTGRN", //ssn
    new Person("Robert", "Green") );

Person bob = people.get("RBTGRN");
if( bob == null )
    System.out.println( "Not found" );

int populationSize = people.size();
```

Generic collections

- Since Java 5, all collection interfaces and classes have been redefined as Generics
- Use of generics leads to code that is
 - ♦ safer
 - ♦ more compact
 - ♦ easier to understand
 - ♦ equally performing

Generic list – excerpt

```
public interface List<E>{
    void add(E x);
    Iterator<E> iterator();
}
public interface Iterator<E>{
    E next();
    boolean hasNext();
}
```

Example

▪ Using a list of Integers

◆ Without generics (`ArrayList list`)

```
list.add(0, new Integer(42));  
int n= ((Integer) (list.get(0))).intValue();
```

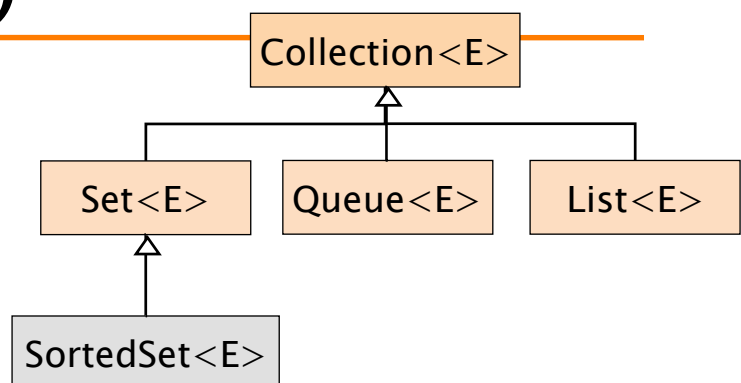
◆ With generics (`ArrayList<Integer> list`)

```
list.add(0, new Integer(42));  
int n= ((Integer) (list.get(0))).intValue();
```

◆ + autoboxing (`ArrayList<Integer> list`)

```
list.add(0, new Integer(42));  
int total = list.get(0).intValue();
```

Group containers (Collections)



List

- Can contain **duplicate** elements
- **Insertion order** is preserved
- User can define insertion point
- Elements can be accessed by **position**
- Augments Collection interface

List specific methods

- **E** **get**(int **index**)
- **E** **set**(int **index**, **E** element)
- void **add**(int **index**, **E** element)
- **E** **remove**(int **index**)

- boolean **addAll**(int **index**, Collection<**E**> c)
- int **indexOf**(**E** o)
- int **lastIndexOf**(**E** o)
- List<**E**> **subList**(int from, int to)

List implementations

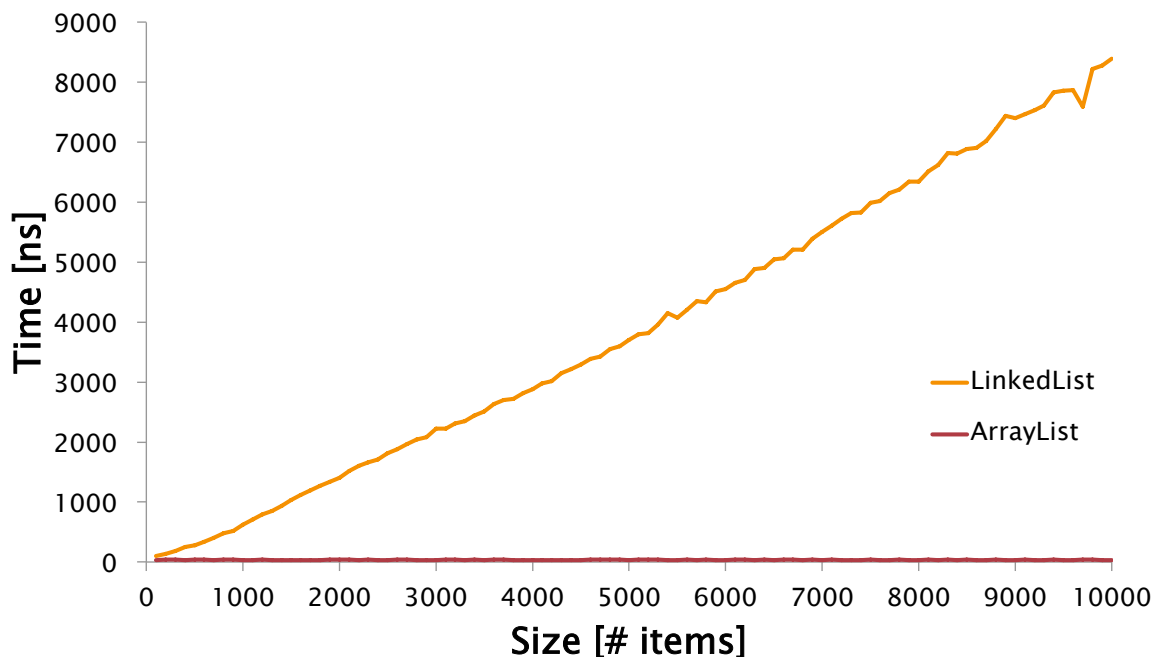
ArrayList

- `get (n)`
 - ♦ Constant
- `add (0 , ...)`
 - ♦ Linear
- `add ()`
 - ♦ Constant

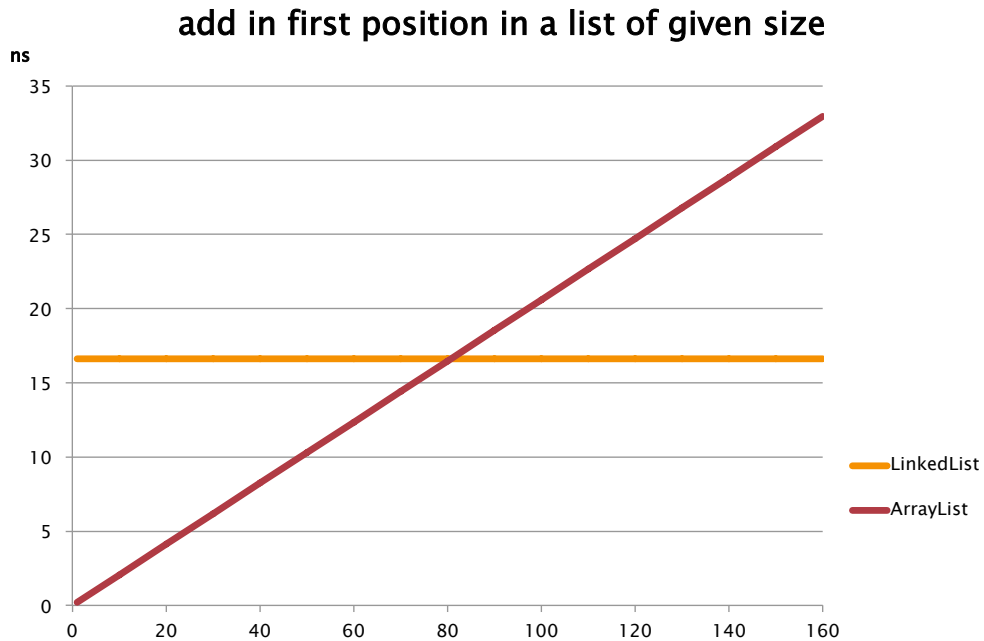
LinkedList

- `get (n)`
 - ♦ Linear
- `add (0 , ...)`
 - ♦ Constant
- `add ()`
 - ♦ Constant

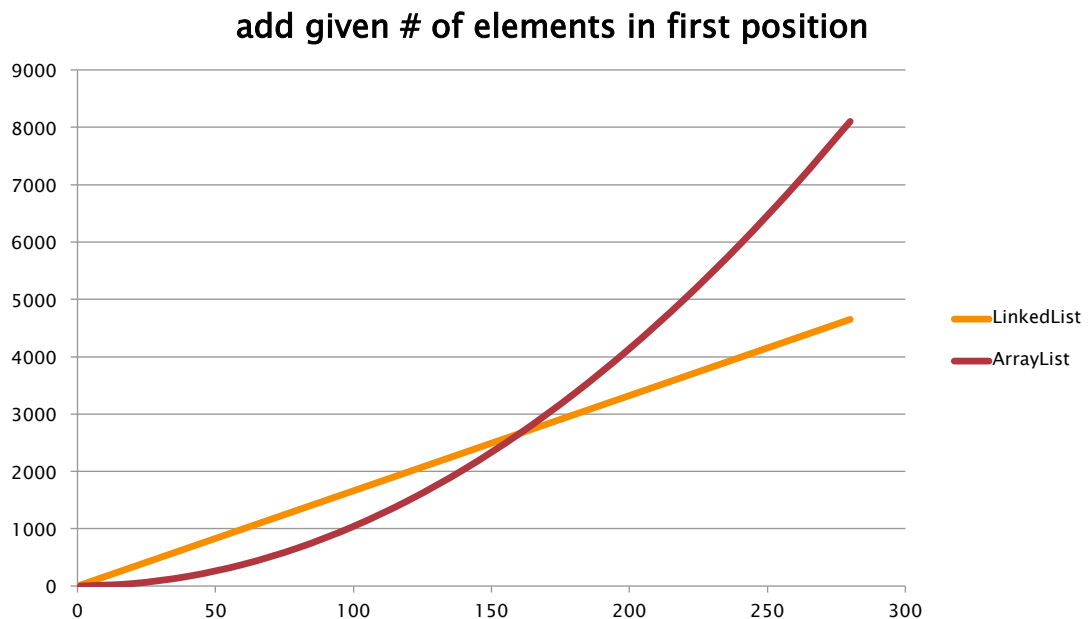
List implementations – Get



List Implementations – Add



List Implementations – Add



List implementation – Models

	LinkedList	ArrayList
Add in first pos. in list of size n	$t(n) = C_L$	$t(n) = n \cdot C_A$
Add n elements	$t(n) = n \cdot C_L$	$t(n) = \sum_{i=1}^n C_A \cdot i$ $= \frac{C_A}{2} n \cdot (n - 1)$
	$C_L = 16.0 \text{ ns}$	
	$C_A = 0.2 \text{ ns}$	

SoftEng
http://softeng.polito.it

List implementations

- **ArrayList**
 - ♦ `ArrayList()`
 - ♦ `ArrayList(int initialCapacity)`
 - ♦ `ArrayList(Collection c)`
 - ♦ `void ensureCapacity(int minCapacity)`
- **LinkedList**
 - ♦ `void addFirst(Object o)`
 - ♦ `void addLast(Object o)`
 - ♦ `Object getFirst()`
 - ♦ `Object getLast()`
 - ♦ `Object removeFirst()`
 - ♦ `Object removeLast()`

SoftEng
http://softeng.polito.it

Example I

```
LinkedList<Integer> l1 =
    new LinkedList<Integer>();

l1.add(new Integer(10));
l1.add(new Integer(11));

l1.addLast(new Integer(13));
l1.addFirst(new Integer(20));

//20, 10, 11, 13
```

Example II

```
Car[] garage = new Car[20];

garage[0] = new Car();
garage[1] = new ElectricCar();
garage[2] =
garage[3] = List<Car> garage = new ArrayList<Car>(20);

for(int i=0; garage.set( 0, new Car() );
    garage[i] garage.set( 1, new ElectricCar() );
}             garage.set( 2, new ElectricCar() );
              garage.set( 3, new Car() );

              for(int i; i<garage.size(); i++){
                  Car c = garage.get(i);
                  c.turnOn();
              }
```


Example III

```
List l = new ArrayList(2); // 2 refs to null

l.add(new Integer(11)); // 11 in position 0
l.add(0, new Integer(13)); // 11 in position 1
l.set(0, new Integer(20)); // 13 replaced by 20

l.add(9, new Integer(30)); // NO: out of bounds
l.add(new Integer(30)); // OK, size extended
```

Queue

- Collection whose elements have an order
 - ◆ not an ordered collection though
- Defines a **head** position where is the **first** element that can be accessed
 - ◆ `peek()`
 - ◆ `poll()`

Queue implementations

- LinkedList
 - ◆ head is the first element of the list
 - ◆ FIFO: First-In-First-Out
- PriorityQueue
 - ◆ head is the smallest element

Queue example

```
Queue<Integer> fifo =  
    new LinkedList<Integer>();  
  
Queue<Integer> pq =  
    new PriorityQueue<Integer>();  
  
fifo.add(3); pq.add(3);  
fifo.add(1); pq.add(1);  
fifo.add(2); pq.add(2);  
  
System.out.println(fifo.peek()); // 3  
System.out.println(pq.peek()); // 1
```

Set

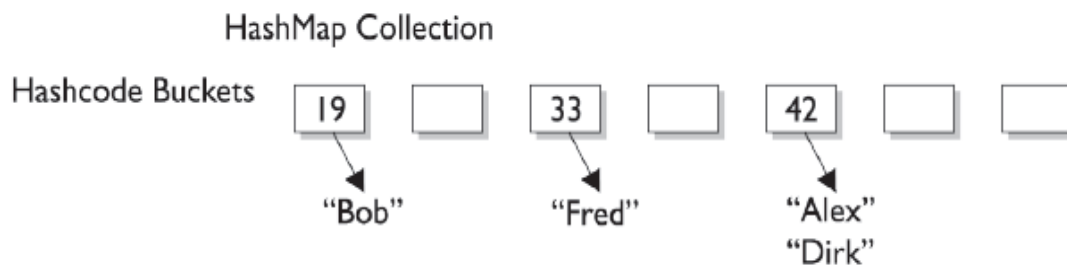
- Contains no methods other than those inherited from Collection
- `add()` has restriction that **no duplicate elements** are allowed
 - ♦ `e1.equals(e2) == false` $\forall e1, e2 \in \Sigma$
- Iterator
 - ♦ The elements are traversed in **no particular order**

The equals() Contract

- It is **reflexive**: `x.equals(x) == true`
- It is **symmetric**: `x.equals(y) == y.equals(x)`
- It is **transitive**: for any reference values `x`, `y`, and `z`,
if `x.equals(y) == true` AND `y.equals(z) == true`
 \Rightarrow `x.equals(z) == true`
- It is **consistent**: for any reference values `x` and `y`,
multiple invocations of `x.equals(y)` consistently return
true (or false), provided that no information used in
equals comparisons on the object is modified.
- `x.equals(null) == false`

hashCode

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	$F(6) + R(18) + E(5) + (D)$	= 33



The hashCode() contract

- The hashCode() method must **consistently** return the same value, if no information used in equals() comparisons on the object is modified.
- If two objects are equal for equals() method, then calling the hashCode() method on the two objects must produce the same integer result.
- If two objects are unequal for equals() method, then calling the hashCode() method on the two objects **MAY** produce distinct integer results.
 - ♦ producing distinct results for unequal objects may improve the performance of hash tables

HashCode()

Condition	Required	Not Required (But Allowed)
<code>x.equals(y) == true</code>	<code>x.hashCode() == y.hashCode()</code>	
<code>x.hashCode() == y.hashCode()</code>		<code>x.equals(y) == true</code>
<code>x.equals(y) == false</code>		No hashCode() requirements
<code>x.hashCode() != y.hashCode()</code>	<code>x.equals(y) == false</code>	

equals() and hashCode()

- equals() and hashCode() are bound together by a joint contract that specifies if two objects are considered equal using the equals() method, then they must have identical hashCode() values.
- To be truly safe:
 - ◆ If override equals(), override hashCode()
 - ◆ Objects that are equals have to return identical hashcodes.

SortedSet

- **No duplicate elements**
- Iterator
 - ◆ The elements are traversed according to the **natural ordering** (ascending)
- Augments Set interface
 - ◆ `Object first()`
 - ◆ `Object last()`
 - ◆ `SortedSet headSet(Object toElement)`
 - ◆ `SortedSet tailSet(Object fromElement)`
 - ◆ `SortedSet subSet(Object from, Object to)`

Set implementations

- **HashSet** implements **Set**
 - ◆ Hash tables as internal data structure (faster)
- **LinkedHashSet** extends HashSet
 - ◆ Elements are traversed by iterator according to the **insertion order**
- **TreeSet** implements **SortedSet**
 - ◆ R-B trees as internal data structure (computationally expensive)

Note on sorted collections

- Depending on the constructor used they require different implementation of the custom ordering
- **TreeSet()**
 - ◆ Natural ordering (elements must be implementations of Comparable)
- **TreeSet(Comparator c)**
 - ◆ Ordering is according to the comparator rules, instead of natural ordering

Iterators



Note well

- It is **unsafe** to iterate over a collection you are modifying (**add/del**) at the same time
- **Unless** you are using the iterator methods
 - ◆ `Iterator.remove()`
 - ◆ `ListIterator.add()`

Delete

```
List<Integer> lst=new LinkedList<Integer>();
lst.add(new Integer(10));
lst.add(new Integer(11));
lst.add(new Integer(13));
lst.add(new Integer(20));

int count = 0;
for (Iterator<?> itr = lst.iterator();
      itr.hasNext(); ) {
    itr.next();
    if (count==1)
        lst.remove(count); // wrong
    count++;
}
```

ConcurrentModificationException

Delete (cont' d)

```
List<Integer> lst=new LinkedList<Integer>();
lst.add(new Integer(10));
lst.add(new Integer(11));
lst.add(new Integer(13));
lst.add(new Integer(20));

int count = 0;
for (Iterator<?> itr = lst.iterator();
     itr.hasNext(); ) {
    itr.next();
    if (count==1)
        itr.remove(); // ok
    count++;
}
```

Correct

Add

```
List lst = new LinkedList();
lst.add(new Integer(10));
lst.add(new Integer(11));
lst.add(new Integer(13));
lst.add(new Integer(20));

int count = 0;
for (Iterator itr = lst.iterator();
     itr.hasNext(); ) {
    itr.next();
    if (count==2)
        lst.add(count, new Integer(22)); //wrong
    count++;
}
```

ConcurrentModificationException

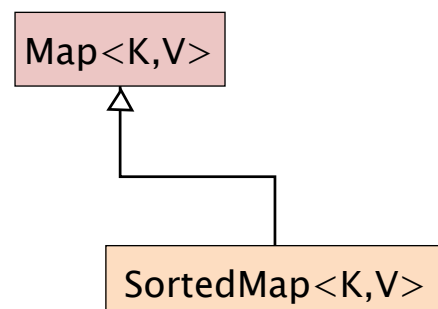
Add (cont' d)

```
List<Integer> lst=new LinkedList<Integer>();
lst.add(new Integer(10));
lst.add(new Integer(11));
lst.add(new Integer(13));
lst.add(new Integer(20));

int count = 0;
for (ListIterator<Integer> itr =
    lst.listIterator(); itr.hasNext();){
    itr.next();
    if (count==2)
        itr.add(new Integer(22)); // ok
    count++;
}
```

Correct

Associative containers (Maps)



Map interface

- `V put(K key, V value)`
- `V get(K key)`
- `Object remove(K key)`
- `boolean containsKey(K key)`
- `boolean containsValue(V value)`
- `public Set<K> keySet()`
- `public Collection<V> values()`
- `int size()`
- `boolean isEmpty()`
- `void clear()`

SortedMap

- The elements are traversed according to the keys' **natural ordering** (ascending)
- Augments Map interface
 - ◆ `SortedMap subMap(K fromKey, K toKey)`
 - ◆ `SortedMap headMap(K toKey)`
 - ◆ `SortedMap tailMap(K fromKey)`
 - ◆ `K firstKey()`
 - ◆ `K lastKey()`

Map implementations

- Analogous to Set
- **HashMap** implements Map
 - ◆ No order
- **LinkedHashMap** extends HashMap
 - ◆ Insertion order
- **TreeMap** implements SortedMap
 - ◆ Ascending key order

HashMap

- Get/put takes **constant time** (in case of no collisions)
- Automatic re-allocation when load factor reached
- Constructor optional arguments
 - ◆ **load factor** (default = .75)
 - ◆ **initial capacity** (default = 16)

Using HashMap

```
Map<String,Student> students =
    new HashMap<String,Student>();

students.put("123",
    new Student("123","Joe Smith"));

Student s = students.get("123");

for(Student si: students.values()){

}
```

Objects Sorting



Comparable interface

```
public interface Comparable<T> {  
    public int compareTo(T obj);  
}
```

- Compares the receiving object with the specified object.
- Return value must be:
 - ◆ <0 if *this* precedes *obj*
 - ◆ $=0$ if *this* has the same order as *obj*
 - ◆ >0 if *this* follows *obj*

Comparable

- The interface is implemented by language common types in packages `java.lang` and `java.util`
 - ◆ String objects are lexicographically ordered
 - ◆ Date objects are chronologically ordered
 - ◆ Number and sub-classes are ordered numerically

Custom ordering

- How to define an ordering upon **Student** objects according to the “natural alphabetic order”

```
public class Student
    implements Comparable<Student>{
    private String first;
    private String last;
    public int compareTo(Student o){
        ...
    }
}
```

Custom ordering

```
public int compareTo(Student o){

    int cmp = lastName.compareTo(s.lastName);

    if(cmp!=0)
        return cmp;
    else
        return firstName.compareTo(s.firstName);
}
```

Ordering “the old way”

- In pre Java 5 code we had:
 - ◆ `public int compareTo(Object obj)`
- No control on types
- A cast had to be performed within the method
 - ◆ Possible `ClassCastException` when comparing objects of unrelated types

Ordering “the old way”

```
public int compareTo(Object obj) {  
    Student s = (Student) obj;  
    int cmp = lastName.compareTo(s.lastName);  
    if (cmp != 0)  
        return cmp;  
    else  
        return firstName.compareTo(s.firstName);  
}
```

possible run-time error

Custom ordering (alternative)

```
public interface Comparator<T> {  
    public int compare(T o1, T o2);  
}
```

- `java.util`
- Compares its two arguments
- Return value must be
 - ◆ <0 if `o1` precedes `o2`
 - ◆ $=0$ if `o1` has the same ordering as `o2`
 - ◆ >0 if `o1` follows `o2`

SoftEng
<http://softeng.polito.it>

65

Custom ordering (alternative)

```
class StudentIDComparator  
    implements Comparator<Student> {  
    public int compare(Student s1, Student s2) {  
        return s1.getID() - s2.getID();  
    }  
}
```

- Usually used to define alternative orderings to Comparable
- The “old way” version compares two Object references

SoftEng
<http://softeng.polito.it>

66

Algorithms



Algorithms

- Static methods of [java.util.Collections](#) class
 - ♦ Work on lists
- `sort()` – merge sort, $n \log(n)$
- `binarySearch()` – requires ordered sequence
- `shuffle()` – unsort
- `reverse()` – requires ordered sequence
- `rotate()` – of given a distance
- `min()`, `max()` – in a Collection

Sort method

- Two generic overloads:

- ♦ on Comparable objects:

```
public static <T extends Comparable<? super T>>  
void sort(List<T> list)
```

- ♦ using a Comparator object:

```
public static <T>  
void sort(List<T> list, Comparator<? super T>)
```

Sort generic

```
T extends Comparable<? super T>  
MasterStudent      Student      MasterStudent
```

- Why <? super T> instead of just <T> ?

- ♦ Suppose you define

- MasterStudent extends Student { }

- ♦ Intending to inherit the Student ordering

- It does not implement

- Comparable<MasterStudent>

- But MasterStudent extends (indirectly)

- Comparable<Student>

Custom ordering (alternative)

```
List students = new LinkedList();

students.add(new Student("Mary", "Smith", 34621));
students.add(new Student("Alice", "Knight", 13985));
students.add(new Student("Joe", "Smith", 95635));

Collections.sort(students); // sort by name

Collections.sort(students,
    new StudentIDComparator()); // sort by ID
```

Search

- `<T> int binarySearch(List<? extends Comparable<? super T>> l, T key)`
 - ◆ Searches the specified object
 - ◆ List must be sorted into ascending order according to natural ordering
- `<T> int binarySearch(List<? extends T> l, T key, Comparator<? super T> c)`
 - ◆ Searches the specified object
 - ◆ List must be sorted into ascending order according to the specified comparator

Algorithms – Arrays

- Static methods of `java.util.Arrays` class
 - ♦ Work on object arrays
- `sort()`
- `binarySearch()`

Search – Arrays

- `int binarySearch(Object[] a, Object key)`
 - ♦ Searches the specified object
 - ♦ Array must be sorted into ascending order according to natural ordering
- `int binarySearch(Object[] a, Object key, Comparator c)`
 - ♦ Searches the specified object
 - ♦ Array must be sorted into ascending order according to the specified comparator

License (1)

- THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.
- BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.
- **1. Definitions**
 - **"Collective Work"** means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.
 - **"Derivative Work"** means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-representation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.
 - **"Licensor"** means the individual or entity that offers the Work under the terms of this License.
 - **"Original Author"** means the individual or entity who created the Work.
 - **"Work"** means the copyrightable work of authorship offered under the terms of this License.
 - **"You"** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.
- **2. Fair Use Rights.** Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.
- **3. License Grant.** Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:
 - a. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;
 - b. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats, but otherwise you have no rights to make Derivative Works. All rights not expressly granted by Licensor are hereby reserved, including but not limited to the rights set forth in Sections 4(d) and 4(e).

/ 5

License (2)

- **4. Restrictions.** The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:
 - a. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any credit as required by clause 4(c), as requested.
 - b. You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.
 - c. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work, You must keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or (ii) if the Original Author and/or Licensor designate another party or parties (e.g. a sponsor institute, publishing entity, journal) for attribution in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; the title of the Work if supplied; and to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work. Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.
 - d. For the avoidance of doubt, where the Work is a musical composition:
 - i. **Performance Royalties Under Blanket Licenses.** Licensor reserves the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work if that performance is primarily intended for or directed toward commercial advantage or private monetary compensation.
 - ii. **Mechanical Rights and Statutory Royalties.** Licensor reserves the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions), if Your distribution of such cover version is primarily intended for or directed toward commercial advantage or private monetary compensation.
 - **Webcasting Rights and Statutory Royalties.** For the avoidance of doubt, where the Work is a sound recording, Licensor reserves the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions), if Your public digital performance is primarily intended for or directed toward commercial advantage or private monetary compensation.

License (3)

- **5. Representations, Warranties and Disclaimer**
 - UNLESS OTHERWISE MUTUALLY AGREED BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.
 - **6. Limitation on Liability.** EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
 - **7. Termination**
 - a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.
 - b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.
 - **8. Miscellaneous**
 - a. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.
 - b. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.
 - c. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.
 - d. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.
-