

Java Collection Framework



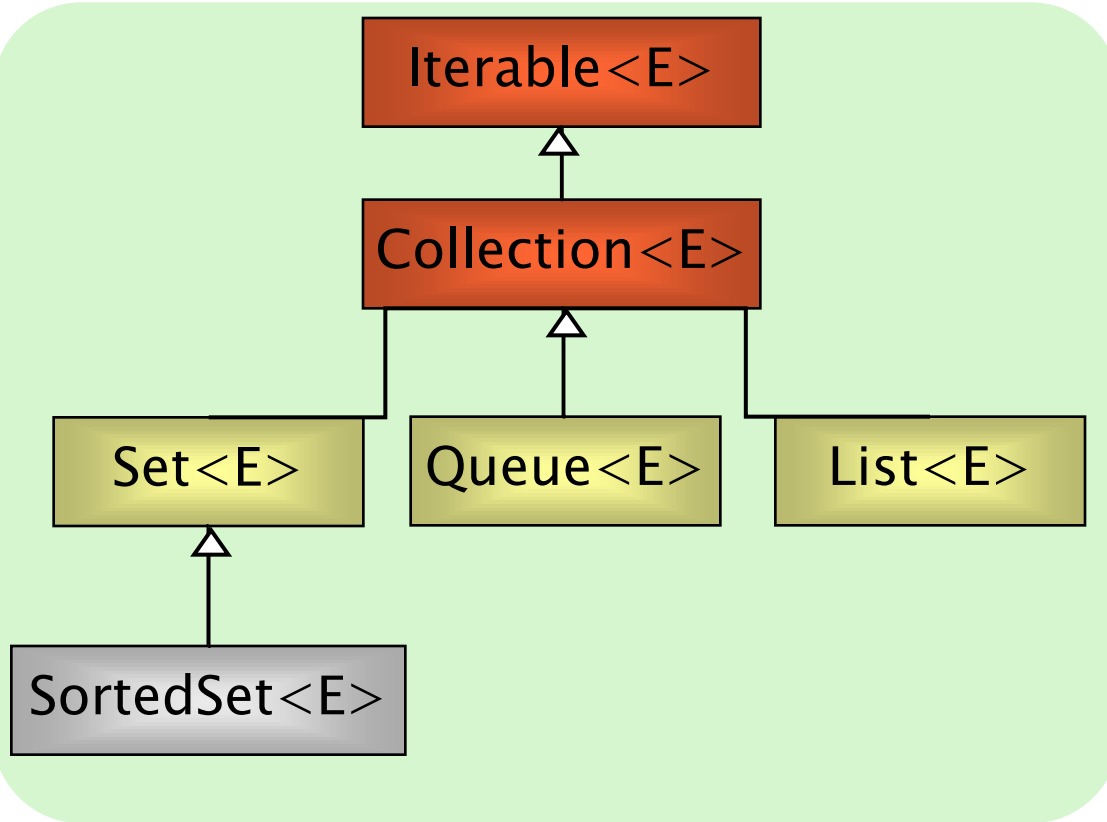
SoftEng
<http://softeng.polito.it>

Version March 2009

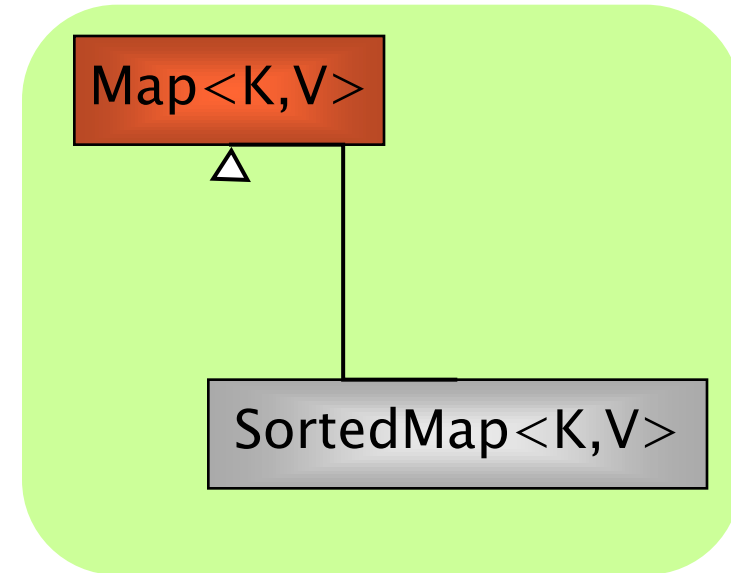
Framework

- Interfaces (ADT, Abstract Data Types)
- Implementations (of ADT)
- Algorithms (sort)
- `java.util.*`
- Java 5 released!
 - ♦ Lots of changes about collections

Interfaces

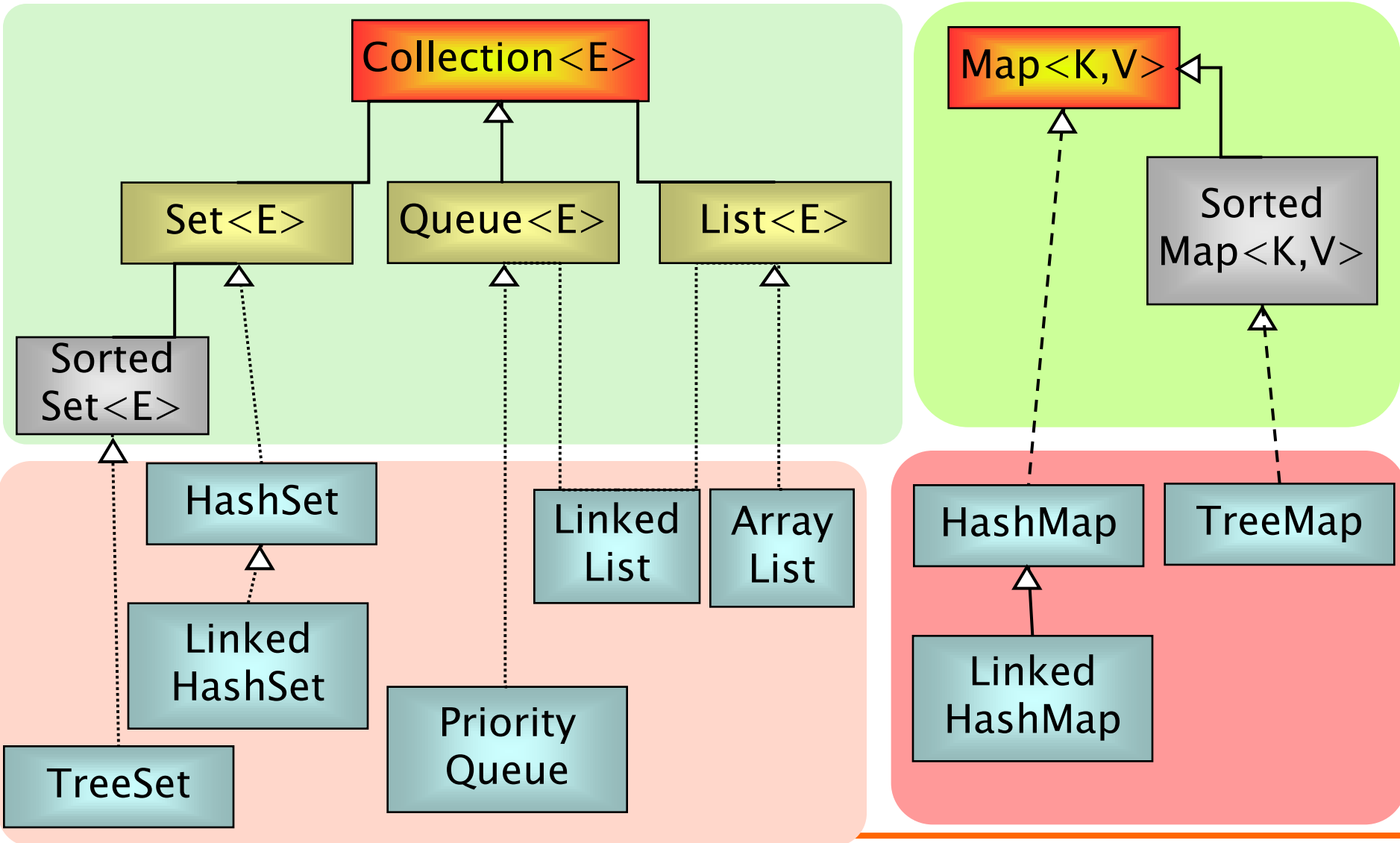


Group containers



Associative containers

Implementations



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

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
 - ♦ T()
 - ♦ T(Collection c)

Collection interface

- `int size()`
- `boolean isEmpty()`
- `boolean contains(Object element)`
- `boolean containsAll(Collection c)`
- `boolean add(Object element)`
- `boolean addAll(Collection c)`
- `boolean remove(Object element)`
- `boolean removeAll(Collection c)`
- `void clear()`
- `Object[] toArray()`
- `Iterator 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
 - ♦ `T ()`
 - ♦ `T (Map m)`

Map interface

- Object **put**(Object key, Object value)
- Object **get**(Object key)
- Object **remove**(Object key)
- boolean **containsKey**(Object key)
- boolean **containsValue**(Object value)
- public Set **keySet**()
- public Collection **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

- From Java 5, all collection interfaces and classes have been redefined as Generics
- Use of generics lead 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();
```

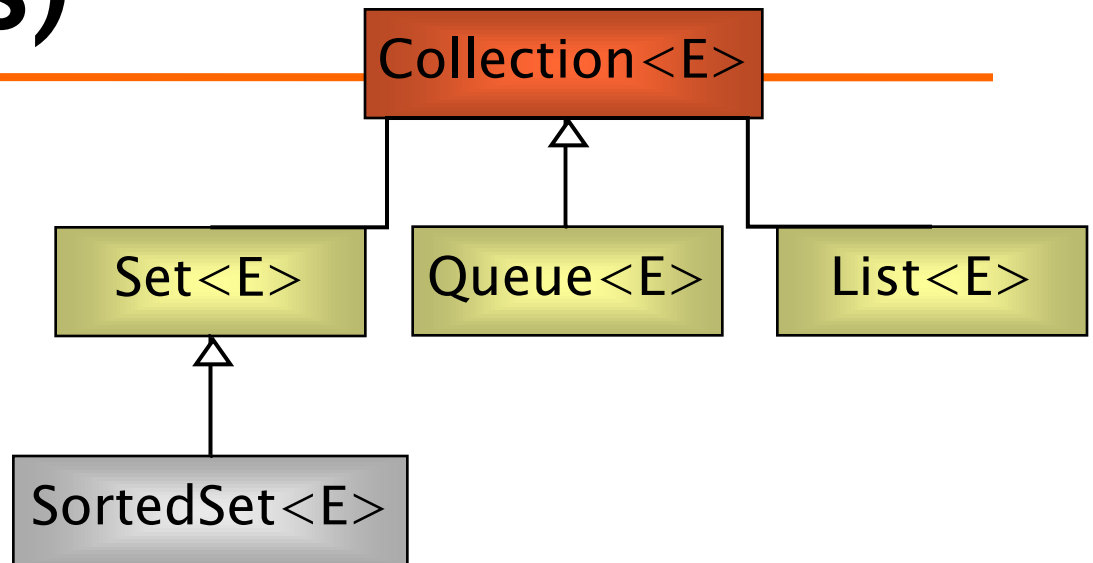
♦ With generics (`ArrayList<Integer> list`)

```
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 additional methods

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

- boolean **addAll**(int **index**, Collection c)
- int **indexOf**(Object o)
- int **lastIndexOf**(Object o)
- List **subList**(int fromIndex, int toIndex)

List implementations

ArrayList

- `get(n)`
 - ♦ Constant time
- Insert (beginning) and delete while iterating
 - ♦ Linear

LinkedList

- `get(n)`
 - ♦ Linear time
- Insert (beginning) and delete while iterating
 - ♦ Constant

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()`

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

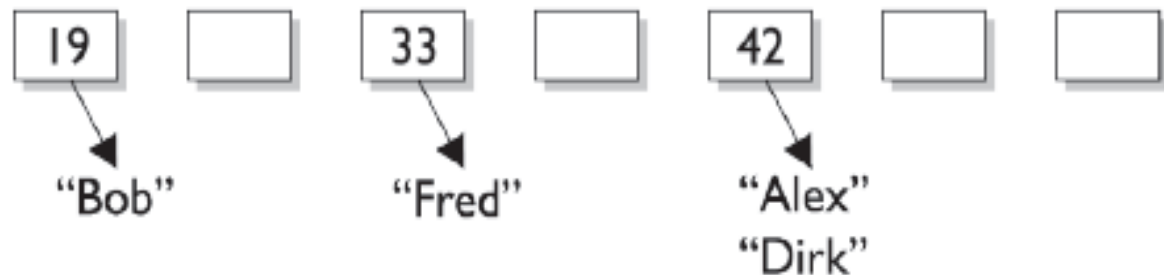
- It is **reflexive**: `x.equals(x) == true`
- It is **symmetric**: `x.equals(y) == y.equals(x)`
- It is **transitive**: for any reference values `x`, `y`, and `z`,
if `x.equals(y) == true` AND `y.equals(z) == true`
 \Rightarrow `x.equals(z) == true`
- It is **consistent**: for any reference values `x` 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	$F(6) + R(18) + E(5) + (D)$	$= 33$

HashMap Collection

Hashcode Buckets



The hashCode() contract

- The hashCode() method must consistently return the same int, 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 int results for unequal objects may improve the performance of hashtables

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 <code>hashCode()</code> 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)`

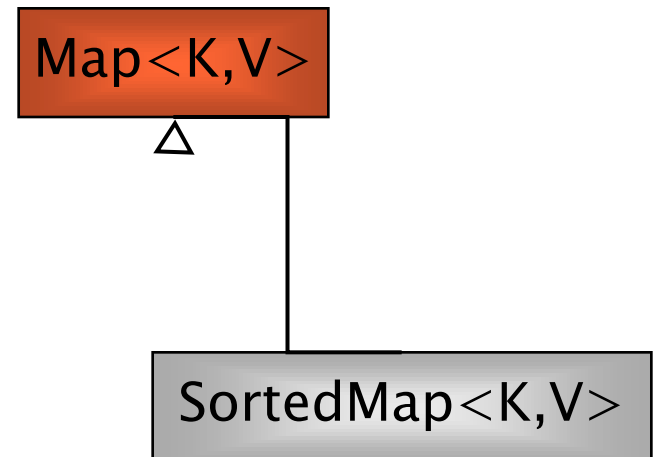
Set implementations

- **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

Associative containers (Maps)



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

HashMap

- Get/set 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()) {  
  
}
```

Iterators



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

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 interface

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

Iterator examples

Print all objects in a list

```
Collection<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

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

Iteration examples

Print all values in a map
(variant using while)

```
Map<String, Person> people =  
    new HashMap<String, Person>();  
...  
Collection<Person> values = people.values();  
  
for(Person p: values) {  
    System.out.println(p);  
}
```

Iteration examples

Print all keys AND values in a map

```
Map<String, Person> people =  
    new HashMap<String, Person>();  
...  
Collection<String> keys = people.keySet();  
for(String ssn: keys) {  
    Person p = people.get(ssn);  
    System.out.println(ssn + " - " + 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();  
    ...  
}
```

Iteration examples (until Java 1.4)

Print all values in a map
(variant using while)

```
Map people = new HashMap();  
...  
Collection values = people.values();  
Iterator i = values.iterator();  
while( i.hasNext() ) {  
    Person p = (Person)i.next();  
    ...  
}
```


Iteration examples (until Java 1.4)

Print all keys AND values in a map

```
Map people = new HashMap();  
...  
Collection keys = people.keySet();  
for(Iterator i= keys.iterator(); i.hasNext(); ) {  
    String ssn = (String)i.next();  
    Person p = (Person)people.get(ssn);  
    ...  
}
```

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

Objects Ordering



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

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

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

Algorithms



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

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