**Collections**

The most important Interfaces and Classes:

*<<Interface>>***Collection**

*<<Interface>>***Set**

*<<Interface>>***List**

*<<Interface>>***Queue**

*<<Interface>>****S*ortedSet**

*<<Interface>>***NavigableSet**

**LinkedList**

**HashSet**

**LinkedHashSet**

**TreeSet**

**ArrayList**

**PriorityQueue**

**Vector**

*<<Interface>>***Dequeue**

*<<Interface>>***Map**

*<<Interface>>***SortedMap**

*<<Interface>>***NavigableMap**

**TreeMap**

**HashMap**

**LinkedHashMap**

**Hashtable**

|  |  |  |
| --- | --- | --- |
| **Collection Class** | **Features** | **Sorting** |
| Vector | - ordered (element access via index)  - not sorted  - doublets are allowed  - Thread-safe  - important Methods:  *add, get, size, contains, indexOf, iterator, toArray, remove*  - Legacy-Collection: should not be used in future (use instead *ArrayList*) | Collections.sort(list)  Collections.sort(list, comp) |
| ArrayList | - ordered (element access via index)  - not sorted  - doublets are allowed  - not Thread-safe  - important Methods:  *add, get, size, contains, indexOf, iterator, toArray, remove* | Collections.sort(list)  Collections.sort(list, comp) |
| LinkedList | - elements are linkt to each other (insertion of an element at a given point is much more faster than with ArrayList)  - ordered (element access via index)  - not sorted  - doublets are allowed  - not Thread-safe  - important Methods:  *add, get, size, contains, indexOf, iterator, toArray, remove, peek, poll, offer* | Collections.sort(list)  Collections.sort(list, comp) |
| PriorityQueue | - ordered  - Sorted: by the natural order of the elements (By the means of a Comparator the order may be changed)  - not Thread-safe  - null-Elements are not allowed  - Important methods:  *peek, poll, offer, iterator, remove, size* |  |
| HashSet | - No doublets will be accepted  - not ordered (random access to elements if iterating through the Set)  - not sorted  - hashCode() and equals() methods of the Element type should be overridden  - very effective, fast access in case of large data amounts  - not thread-safe  - Important methods:  *add, contains, isEmpty, iterator, remove, size* | new TreeSet(set)  oder  new TreeSet(HashSet set, Comparator comp) |
| LinkedHashSet | Same as HashSet but ordered.  If iterating trough the set, access to the elements is performed with the original insertion order ( not random as it is with HashSet) | new TreeSet(set)  oder  new TreeSet(set,comp) |

|  |  |  |
| --- | --- | --- |
| TreeSet | Same like HashSet but:  - sorted (natural order of the element type or defined by comparator)  - the Interfaces : *Comparable or Comparator* must be implemented  - iteration over the elements in a sorted order  - Java 6 provides the Interface *NavigableSet.* |  |
| HashMap | - stored Key/Value-Pairs  - doublets of Keys is not possible (same key is not rejected but the new key/value pair overrides the old one)  - doublets of Values are possible  - not ordered (concerning keys)  - not sorted  - very effective, fast access in case of large data amounts  - not thread-safe  - Important methods:  *put, get, keySet, entrySet, values, remove, size, containsKey, containsValue* | new TreeMap(map)  oder  new TreeMap(map,comp) |
| LinkedHashMap | Same as HashMap, but: - ordered (iterating and accessing keys is not random but same as the insertion order) | new TreeMap(map)  oder  new TreeMap(map,comp) |
| TreeMap | Same like HashMap but:  - sorted (natural order of the key type or defined by a comparator)  - the Interfaces : *Comparable or Comparator* must be implemented  - Java Version 6 provides the Interface *NavigableMap.* |  |

**Summary of the most important methods in collections**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Important methods** | **List** | **Set** | **Map** | **Queue** | **Comments** |
| boolean add(e) boolean add(index,e) | **x x** | **x** |  |  | add a new element |
| boolean contains(obj) boolean containsKey(key) boolean containsValue(value) | **x** | **x** | **x x** | **x** | checks if Object is Element of collection |
| Object get(index) Object get(key) | **x** |  | **x** |  | read an Element |
| int indexOf(obj) | **x** |  |  |  | returns the position of the Element |
| Iterator iterator() | **x** | **x** |  | **x** | returns an Iterator |
| Set keySet() Set entrySet() Collection values() |  |  | **x x x** |  | Set- und Collection Views of a Map |
| put(key,value) |  |  | **x** |  | add a Key-Value Pair |
| remove(index) remove(obj) remove(key) | **x x** | **x** | **x** |  | remove an Element |
| int size() | **x** | **x** | **x** | **x** | count of Elements |
| Object[] toArray() T[] toArray(T[]) | **x x** | **x x** |  |  | Creates an array from the collection |
| offer(e) returns: **false** if no success add(e) exception if no success |  |  |  | **x x** | add head element |
| peek() returns: **null** if empty  element() exception if empty |  |  |  | **x x** | read head-element (1.element) |
| poll() returns: **null** if empty  remove() exception if empty |  |  |  | **x x** | read and remove head-element (1.element) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Summary of Deque methods | | | | |
|  | **First Element (Head)** | | **Last Element (Tail)** | |
|  | *Throws exception* | *Special value* | *Throws exception* | *Special value* |
| **Insert** | [addFirst(e)](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#addFirst-E-) | [offerFirst(e)](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#offerFirst-E-) | [addLast(e)](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#addLast-E-) | [offerLast(e)](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#offerLast-E-) |
| **Remove** | [removeFirst()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#removeFirst--) | [pollFirst()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#pollFirst--) | [removeLast()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#removeLast--) | [pollLast()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#pollLast--) |
| **Examine** | [getFirst()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#getFirst--) | [peekFirst()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#peekFirst--) | [getLast()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#getLast--) | [peekLast()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#peekLast--) |

* This interface extends the [Queue](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Queue.html) interface. When a deque is used as a queue, FIFO (First-In-First-Out) behavior results. Elements are added at the end of the deque and removed from the beginning. The methods inherited from the Queue interface are precisely equivalent to Deque methods as indicated in the following table:

|  |  |
| --- | --- |
| Comparison of Queue and Deque methods | |
| **Queue Method** | **Equivalent Deque Method** |
| [add(e)](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Queue.html#add-E-) | [addLast(e)](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#addLast-E-) |
| [offer(e)](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Queue.html#offer-E-) | [offerLast(e)](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#offerLast-E-) |
| [remove()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Queue.html#remove--) | [removeFirst()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#removeFirst--) |
| [poll()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Queue.html#poll--) | [pollFirst()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#pollFirst--) |
| [element()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Queue.html#element--) | [getFirst()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#getFirst--) |
| [peek()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Queue.html#peek--) | [peekFirst()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#peek--) |

* Deques can also be used as LIFO (Last-In-First-Out) stacks. This interface should be used in preference to the legacy [Stack](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Stack.html) class. When a deque is used as a stack, elements are pushed and popped from the beginning of the deque. Stack methods are precisely equivalent to Deque methods as indicated in the table below:

|  |  |
| --- | --- |
| Comparison of Stack and Deque methods | |
| **Stack Method** | **Equivalent Deque Method** |
| [push(e)](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#push-E-) | [addFirst(e)](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#addFirst-E-) |
| [pop()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#pop--) | [removeFirst()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#removeFirst--) |
| [peek()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#peek--) | [peekFirst()](mk:@MSITStore:C:\Users\sando\Desktop\j2se8.chm::/j2se8/api/java/util/Deque.html#peekFirst--) |

**Sorting Collections**

1. applying the natural order of objects:

the natural order of objects has to be implemented inside the compareTo(obj) method provided by the Comparable interface.

“It compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.” (JDK 7.0 doc.)  
  
Example: sorting a list of Person objects by their names  
  
class Person implements Comparable<Person> {

String name = "";  
 int age;

Person(String name, int age) {  
 this.age = age;  
 this.name = name;  
 }

@Override  
 public int **compareTo**(Person p) {  
 return this.name.compareTo(p.name);  
 }

The compareTo method of the String class

@Override  
 public String toString(){  
 return name+" \t| age: "+ age;  
 }  
}  
  
The Client application:

public class SortPersons {

static List<Person> persons = new ArrayList();

static public void main(String... s) {  
 persons.add(new Person("Sandor", 35));  
 persons.add(new Person("Anton", 41));  
 persons.add(new Person("Jahn", 25));  
 persons.add(new Person("Lars", 16));  
 persons.add(new Person("Malte", 23));

**Collections.sort(persons);**

for(Person p : persons) Output:  
 System.out.println(p.toString());  
 }  
}

Anton | age: 41  
Jahn | age: 25  
Lars | age: 16  
Malte | age: 23  
Sandor | age: 35

In order to sort by age we just change the compareTo method as follows:

public int compareTo(Person p) {  
 return this.age - p.age;  
}

Output:

Lars | age: 16  
Malte | age: 23  
Jahn | age: 25  
Sandor | age: 35  
Anton | age: 41

In many cases you cannot override or even implement the compareTo method (i.e.: if a class is final lik the String class) or you simply want to dinamically replace the sorting algorythm at runtime.   
Or you need an ordering for collections of objects that don't have a natural ordering.  
In all these cases we have to implement the **comparator** Interface and create our own comparator:

public class MyComparator implements Comparator<Person> {  
  
 private int mode = 0;  
 MyComparator(int mode) {  
 this.mode = mode;  
 }

@Override  
 public int compare(Person p1, Person p2) {

switch(**mode**) {  
 *//sort by the lenth of name* case 0: return p1.name.length() - p2.name.length();   
  
 *//sort alphabetically*  
 case **1**: return p1.name.compareTo(p2.name);   
   
 *//sort by age* case 2: return p1.age - p2.age;   
   
 *// this will not sort at all*  
 default: return 0;  
 }  
 }  
}   
  
Client code to sort apphabetically:  
**Collections.sort(persons, new MyComparator(1));**

Client code to sort by age:  
**Collections.sort(persons, new MyComparator(2));Using Sets**

A Set is a collection of doubletless objects.   
Any inserted object will be rejected if a *meaningfully equal* object is already contained within the set. Basically finding a meaningfully equal object within a big set say 500.000 instances takes about the same time as finding an object within a small set of only a few objects.   
This is guarranted due to the hash principal:

int hashCode()

Bucket\_1

Bucket\_2

Bucket\_n

boolean equals(obj)

Objects intended to be stored in a set need to override the hashCode() and equals() methods, both inherited from the Object class.  
The equals() method: compares objects by their meeningful attribute(s) for example the name attribute of a Person object or the name- and age-attributes, or whatever attribute(s) we desided as relevant for the comparison.   
The hashCode() method generates an integer derived by the same attribute(s) already applied for the equals method.

How does it work:   
objects will be delegated into buckets depending on the hash code provided by the hashCode() method which will be called by the set on each object. Depending on the quality of hash code calculation, ideally only a few objects will fall into the same bucket. Inside a bucket the equals() method is responsible to find a meeningfully equal object enabling the set to reject that object.  
In case of a badly calculated hash code lot of objects will fall into the same bucket and so, the equals method will have to do a lot of work to compare all objects with the inserted one.   
Thus the performance may be reduced drastically!  
Therefore the hash code should be as good as possible providing for different objects different hash codes. For example the String class implements the following hash code calculation:

s[0]\*31^(n-1) + s[1]\*31^(n-2) + ... + s[n-1]

Different strings will have different hash codes as far as possible.  
However it cannot be avoided that two different objects generate the same hash code even with above complex algorythm.

Important rules:

1. if two objects are equal (by the equals(Object) method), they must provide the same hash code.
2. If two objects provide the same hash code, they do not necessarily have to be equal
3. both, the equals(Object) and the hashCode() method must refer to the same attributes
4. transient variables must not be used for the hash calculation or equals detection   
   (refer to Serialization)
5. During the execution of a Java application, whenever hashCode() is invoked on the same object it needs to return consistently the same integer.

Example:   
  
class Cat {  
 private String name;

Cat (String name) {  
 this.name = name;  
 }

public String getName() { return name; }

public int **hashCode()** {  
 return this.name.hashCode();   
 }

public boolean **equals(Object o)** {  
 return this.name.equals(((Cat)o).name);   
 }  
}

Client code:

public class HashTest {

static public void main(String... ar) {

Set<Cat> catery = new HashSet<Cat>();  
 catery.add(new Cat("Odile"));  
 catery.add(new Cat("Odile "));  
 catery.add(new Cat("Odile "));  
 catery.add(new Cat("Snot"));  
 catery.add(new Cat("Sooty"));  
 catery.add(new Cat("Ginger"));  
 catery.add(new Cat("Ginger"));  
 catery.add(new Cat("Lioness"));  
 catery.add(new Cat("Bronco"));  
 catery.add(new Cat(""));

for (Cat c : catery) {  
 System.out.println(c.getName());  
 }  
 }  
 }Result:  
1) see, how doublets are removed  
2) objets are stored ramdomly as a set is an unordered collection

Odile  
Bronco  
Snot  
Ginger  
Sooty  
Lioness

we can obtain the original order if we change HashSet to LinkedHashSet

Odile  
Snot  
Sooty  
Ginger  
Lioness  
Bronco