Java - The LinkedList Class

The LinkedList class extends AbstractSequentialList and implements the List interface. It provides a linked-list data structure.

Given below are the constructors supported by the LinkedList class.

|  |  |
| --- | --- |
| **SN** | **Constructors and Description** |
| 1 | **LinkedList( )**  This constructor builds an empty linked list. |
| 2 | **LnkedList(Collection c)**  This constructor builds a linked list that is initialized with the elements of the collection c. |

Apart from the methods inherited from its parent classes, LinkedList defines following methods:

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **void add(int index, Object element)**  Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index > size()). |
| 2 | **boolean add(Object o)**  Appends the specified element to the end of this list. |
| 3 | **boolean addAll(Collection c)**  Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws NullPointerException if the specified collection is null |
| 4 | **boolean addAll(int index, Collection c)**  Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws NullPointerException if the specified collection is null. |
| 5 | **void addFirst(Object o)**  Inserts the given element at the beginning of this list. |
| 6 | **void addLast(Object o)**  Appends the given element to the end of this list. |
| 7 | **void clear()**  Removes all of the elements from this list. |
| 8 | **Object clone()**  Returns a shallow copy of this LinkedList. |
| 9 | **boolean contains(Object o)**  Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)). |
| 10 | **Object get(int index)**  Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index >= size()). |
| 11 | **Object getFirst()**  Returns the first element in this list. Throws NoSuchElementException if this list is empty. |
| 12 | **Object getLast()**  Returns the last element in this list. Throws NoSuchElementException if this list is empty. |
| 13 | **int indexOf(Object o)**  Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. |
| 14 | **int lastIndexOf(Object o)**  Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. |
| 15 | **ListIterator listIterator(int index)**  Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index >= size()). |
| 16 | **Object remove(int index)**  Removes the element at the specified position in this list. Throws NoSuchElementException if this list is empty. |
| 17 | **boolean remove(Object o)**  Removes the first occurrence of the specified element in this list. Throws NoSuchElementException if this list is empty. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index >= size()). |
| 18 | **Object removeFirst()**  Removes and returns the first element from this list. Throws NoSuchElementException if this list is empty. |
| 19 | **Object removeLast()**  Removes and returns the last element from this list. Throws NoSuchElementException if this list is empty. |
| 20 | **Object set(int index, Object element)**  Replaces the element at the specified position in this list with the specified element. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index >= size()). |
| 21 | **int size()**  Returns the number of elements in this list. |
| 22 | **Object[] toArray()**  Returns an array containing all of the elements in this list in the correct order. Throws NullPointerException if the specified array is null. |
| 23 | **Object[] toArray(Object[] a)**  Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array. |

## Example:

The following program illustrates several of the methods supported by LinkedList:

import java.util.\*;

public class LinkedListDemo {

public static void main(String args[]) {

// create a linked list

LinkedList ll = new LinkedList();

// add elements to the linked list

ll.add("F");

ll.add("B");

ll.add("D");

ll.add("E");

ll.add("C");

ll.addLast("Z");

ll.addFirst("A");

ll.add(1, "A2");

System.out.println("Original contents of ll: " + ll);

// remove elements from the linked list

ll.remove("F");

ll.remove(2);

System.out.println("Contents of ll after deletion: "

+ ll);

// remove first and last elements

ll.removeFirst();

ll.removeLast();

System.out.println("ll after deleting first and last: "

+ ll);

// get and set a value

Object val = ll.get(2);

ll.set(2, (String) val + " Changed");

System.out.println("ll after change: " + ll);

}

}

This would produce the following result:

Original contents of ll: [A, A2, F, B, D, E, C, Z]

Contents of ll after deletion: [A, A2, D, E, C, Z]

ll after deleting first and last: [A2, D, E, C]

ll after change: [A2, D, E Changed, C]

# Java - The Stack Class

Stack is a subclass of Vector that implements a standard last-in, first-out stack.

Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

Stack( )

Apart from the methods inherited from its parent class Vector, Stack defines following methods:

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **boolean empty()**  Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements. |
| 2 | **Object peek( )**  Returns the element on the top of the stack, but does not remove it. |
| 3 | **Object pop( )**  Returns the element on the top of the stack, removing it in the process. |
| 4 | **Object push(Object element)**  Pushes element onto the stack. element is also returned. |
| 5 | **int search(Object element)**  Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, .1 is returned. |

## Example:

The following program illustrates several of the methods supported by this collection:

import java.util.\*;

public class StackDemo {

static void showpush(Stack st, int a) {

st.push(new Integer(a));

System.out.println("push(" + a + ")");

System.out.println("stack: " + st);

}

static void showpop(Stack st) {

System.out.print("pop -> ");

Integer a = (Integer) st.pop();

System.out.println(a);

System.out.println("stack: " + st);

}

public static void main(String args[]) {

Stack st = new Stack();

System.out.println("stack: " + st);

showpush(st, 42);

showpush(st, 66);

showpush(st, 99);

showpop(st);

showpop(st);

showpop(st);

try {

showpop(st);

} catch (EmptyStackException e) {

System.out.println("empty stack");

}

}

}

This would produce the following result:

stack: [ ]

push(42)

stack: [42]

push(66)

stack: [42, 66]

push(99)

stack: [42, 66, 99]

pop -> 99

stack: [42, 66]

pop -> 66

stack: [42]

pop -> 42

stack: [ ]

pop -> empty stack

**Queues**

## Interface Queue<E>

* ****Type Parameters:****

E - the type of elements held in this collection

**All Superinterfaces:**

[Collection](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<E>, [Iterable](https://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html" \o "interface in java.lang)<E>

**All Known Subinterfaces:**

[BlockingDeque](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingDeque.html)<E>, [BlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html" \o "interface in java.util.concurrent)<E>, [Deque](https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html" \o "interface in java.util)<E>, [TransferQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TransferQueue.html" \o "interface in java.util.concurrent)<E>

**All Known Implementing Classes:**

[AbstractQueue](https://docs.oracle.com/javase/7/docs/api/java/util/AbstractQueue.html), [ArrayBlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html), [ArrayDeque](https://docs.oracle.com/javase/7/docs/api/java/util/ArrayDeque.html), [ConcurrentLinkedDeque](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedDeque.html), [ConcurrentLinkedQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html), [DelayQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/DelayQueue.html), [LinkedBlockingDeque](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingDeque.html), [LinkedBlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html), [LinkedList](https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html), [LinkedTransferQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedTransferQueue.html), [PriorityBlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/PriorityBlockingQueue.html), [PriorityQueue](https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html),[SynchronousQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/SynchronousQueue.html)

public interface ****Queue<E>****

extends [Collection](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<E>

A collection designed for holding elements prior to processing. Besides basic [Collection](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html) operations, queues provide additional insertion, extraction, and inspection operations. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted Queue implementations; in most implementations, insert operations cannot fail.

|  |  |  |
| --- | --- | --- |
|  | Throws exception | Returns special value |
| **Insert** | [add(e)](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#add(E)) | [offer(e)](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#offer(E)) |
| **Remove** | [remove()](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#remove()) | [poll()](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#poll()) |
| **Examine** | [element()](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#element()) | [peek()](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#peek()) |

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out). Whatever the ordering used, the head of the queue is that element which would be removed by a call to [remove()](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#remove()) or [poll()](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#poll()). In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.

The [offer](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#offer(E)) method inserts an element if possible, otherwise returning false. This differs from the [Collection.add](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html" \l "add(E)) method, which can fail to add an element only by throwing an unchecked exception. The offer method is designed for use when failure is a normal, rather than exceptional occurrence, for example, in fixed-capacity (or "bounded") queues.

The [remove()](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#remove()) and [poll()](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#poll()) methods remove and return the head of the queue. Exactly which element is removed from the queue is a function of the queue's ordering policy, which differs from implementation to implementation. The remove()and poll() methods differ only in their behavior when the queue is empty: the remove() method throws an exception, while the poll() method returns null.

The [element()](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#element()) and [peek()](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#peek()) methods return, but do not remove, the head of the queue.

The Queue interface does not define the *blocking queue methods*, which are common in concurrent programming. These methods, which wait for elements to appear or for space to become available, are defined in the [BlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html" \o "interface in java.util.concurrent)interface, which extends this interface.

Queue implementations generally do not allow insertion of null elements, although some implementations, such as [LinkedList](https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html" \o "class in java.util), do not prohibit insertion of null. Even in the implementations that permit it, null should not be inserted into aQueue, as null is also used as a special return value by the poll method to indicate that the queue contains no elements.

Queue implementations generally do not define element-based versions of methods equals and hashCode but instead inherit the identity based versions from class Object, because element-based equality is not always well-defined for queues with the same elements but different ordering properties.

This interface is a member of the [Java Collections Framework](https://docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html).

****Since:****

1.5

****See Also:****

[Collection](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html), [LinkedList](https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html), [PriorityQueue](https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html), [LinkedBlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html), [BlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html), [ArrayBlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html), [LinkedBlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html), [PriorityBlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/PriorityBlockingQueue.html)

### Method Summary

|  |  |
| --- | --- |
| **Methods** | |
| **Modifier and Type** | **Method and Description** |
| boolean | [**add**](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#add(E))([**E**](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) e)  Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing anIllegalStateException if no space is currently available. |
| [**E**](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) | [**element**](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#element())()  Retrieves, but does not remove, the head of this queue. |
| boolean | [**offer**](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#offer(E))([**E**](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) e)  Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions. |
| [**E**](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) | [**peek**](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#peek())()  Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. |
| [**E**](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) | [**poll**](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#poll())()  Retrieves and removes the head of this queue, or returns null if this queue is empty. |
| [**E**](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) | [**remove**](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#remove())()  Retrieves and removes the head of this queue. |

### Methods inherited from interface java.util.[Collection](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)

[addAll](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#addAll(java.util.Collection)), [clear](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#clear()), [contains](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#contains(java.lang.Object)), [containsAll](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#containsAll(java.util.Collection)), [equals](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#equals(java.lang.Object)), [hashCode](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#hashCode()), [isEmpty](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#isEmpty()), [iterator](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#iterator()), [remove](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#remove(java.lang.Object)), [removeAll](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#removeAll(java.util.Collection)), [retainAll](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#retainAll(java.util.Collection)), [size](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#size()), [toArray](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#toArray()), [toArray](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#toArray(T[]))

### Method Detail

#### add

boolean add([E](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) e)

Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.

**Specified by:**

[add](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#add(E)) in interface [Collection](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<[E](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html)>

****Parameters:****

e - the element to add

****Returns:****

true (as specified by [Collection.add(E)](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html" \l "add(E)))

****Throws:****

[IllegalStateException](https://docs.oracle.com/javase/7/docs/api/java/lang/IllegalStateException.html) - if the element cannot be added at this time due to capacity restrictions

[ClassCastException](https://docs.oracle.com/javase/7/docs/api/java/lang/ClassCastException.html) - if the class of the specified element prevents it from being added to this queue

[NullPointerException](https://docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html) - if the specified element is null and this queue does not permit null elements

[IllegalArgumentException](https://docs.oracle.com/javase/7/docs/api/java/lang/IllegalArgumentException.html) - if some property of this element prevents it from being added to this queue

#### offer

boolean offer([E](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) e)

Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions. When using a capacity-restricted queue, this method is generally preferable to [add(E)](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#add(E)), which can fail to insert an element only by throwing an exception.

****Parameters:****

e - the element to add

****Returns:****

true if the element was added to this queue, else false

****Throws:****

[ClassCastException](https://docs.oracle.com/javase/7/docs/api/java/lang/ClassCastException.html) - if the class of the specified element prevents it from being added to this queue

[NullPointerException](https://docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html) - if the specified element is null and this queue does not permit null elements

[IllegalArgumentException](https://docs.oracle.com/javase/7/docs/api/java/lang/IllegalArgumentException.html) - if some property of this element prevents it from being added to this queue

#### remove

[E](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) remove()

Retrieves and removes the head of this queue. This method differs from [poll](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#poll()) only in that it throws an exception if this queue is empty.

****Returns:****

the head of this queue

****Throws:****

[NoSuchElementException](https://docs.oracle.com/javase/7/docs/api/java/util/NoSuchElementException.html) - if this queue is empty

#### poll

[E](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) poll()

Retrieves and removes the head of this queue, or returns null if this queue is empty.

****Returns:****

the head of this queue, or null if this queue is empty

#### element

[E](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) element()

Retrieves, but does not remove, the head of this queue. This method differs from [peek](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#peek()) only in that it throws an exception if this queue is empty.

****Returns:****

the head of this queue

****Throws:****

[NoSuchElementException](https://docs.oracle.com/javase/7/docs/api/java/util/NoSuchElementException.html) - if this queue is empty

#### peek

[E](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) peek()

Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

****Returns:****

the head of this queue, or null if this queue is empty

**Sets in JAVA**

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction.

The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.

Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ.

The methods declared by Set are summarized in the following table:

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **add( )**  Adds an object to the collection |
| 2 | **clear( )**  Removes all objects from the collection |
| 3 | **contains( )**  Returns true if a specified object is an element within the collection |
| 4 | **isEmpty( )**  Returns true if the collection has no elements |
| 5 | **iterator( )**  Returns an Iterator object for the collection which may be used to retrieve an object |
| 6 | **remove( )**  Removes a specified object from the collection |
| 7 | **size( )**  Returns the number of elements in the collection |

## Example:

Set have its implementation in various classes like HashSet, TreeSet, LinkedHashSet. Following is the example to explain Set functionality:

import java.util.\*;

public class SetDemo {

public static void main(String args[]) {

int count[] = {34, 22,10,60,30,22};

Set<Integer> set = new HashSet<Integer>();

try{

for(int i = 0; i<5; i++){

set.add(count[i]);

}

System.out.println(set);

TreeSet sortedSet = new TreeSet<Integer>(set);

System.out.println("The sorted list is:");

System.out.println(sortedSet);

System.out.println("The First element of the set is: "+

(Integer)sortedSet.first());

System.out.println("The last element of the set is: "+

(Integer)sortedSet.last());

}

catch(Exception e){}

}

}

This would produce the following result:

[amrood]$ java SetDemo

[34, 30, 60, 10, 22]

The sorted list is:

[10, 22, 30, 34, 60]

The First element of the set is: 10

The last element of the set is: 60

# Java - The Map Interface

The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date.

* Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.
* Several methods throw a NoSuchElementException when no items exist in the invoking map.
* A ClassCastException is thrown when an object is incompatible with the elements in a map.
* A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the map.
* An UnsupportedOperationException is thrown when an attempt is made to change an unmodifiable map.

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **void clear( )**  Removes all key/value pairs from the invoking map. |
| 2 | **boolean containsKey(Object k)**  Returns true if the invoking map contains k as a key. Otherwise, returns false. |
| 3 | **boolean containsValue(Object v)**  Returns true if the map contains v as a value. Otherwise, returns false |
| 4 | **Set entrySet( )**  Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. This method provides a set-view of the invoking map. |
| 5 | **boolean equals(Object obj)**  Returns true if obj is a Map and contains the same entries. Otherwise, returns false. |
| 6 | **Object get(Object k)**  Returns the value associated with the key k. |
| 7 | **int hashCode( )**  Returns the hash code for the invoking map. |
| 8 | **boolean isEmpty( )**  Returns true if the invoking map is empty. Otherwise, returns false. |
| 9 | **Set keySet( )**  Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map. |
| 10 | **Object put(Object k, Object v)**  Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned. |
| 11 | **void putAll(Map m)**  Puts all the entries from m into this map. |
| 12 | **Object remove(Object k)**  Removes the entry whose key equals k. |
| 13 | **int size( )**  Returns the number of key/value pairs in the map. |
| 14 | **Collection values( )**  Returns a collection containing the values in the map. This method provides a collection-view of the values in the map. |

## Example:

Map has its implementation in various classes like HashMap. Following is the example to explain map functionality:

import java.util.\*;

public class CollectionsDemo {

public static void main(String[] args) {

Map m1 = new HashMap();

m1.put("Zara", "8");

m1.put("Mahnaz", "31");

m1.put("Ayan", "12");

m1.put("Daisy", "14");

System.out.println();

System.out.println(" Map Elements");

System.out.print("\t" + m1);

}

}

This would produce the following result:

ap Elements

{Mahnaz=31, Ayan=12, Daisy=14, Zara=8}

It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array or an array of any type that supports ordering.

Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods or, with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generic Methods:

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods:

* All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type ( < E > in the next example).
* Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
* The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
* A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Example:

Following example illustrates how we can print array of different type using a single Generic method:

public class GenericMethodTest

{

// generic method printArray

public static < E > void printArray( E[] inputArray )

{

// Display array elements

for ( E element : inputArray ){

System.out.printf( "%s ", element );

}

System.out.println();

}

public static void main( String args[] )

{

// Create arrays of Integer, Double and Character

Integer[] intArray = { 1, 2, 3, 4, 5 };

Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };

Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

System.out.println( "Array integerArray contains:" );

printArray( intArray ); // pass an Integer array

System.out.println( "\nArray doubleArray contains:" );

printArray( doubleArray ); // pass a Double array

System.out.println( "\nArray characterArray contains:" );

printArray( charArray ); // pass a Character array

}

}

This would produce the following result:

Array integerArray contains:

1 2 3 4 5 6

Array doubleArray contains:

1.1 2.2 3.3 4.4

Array characterArray contains:

H E L L O

Bounded Type Parameters:

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

Example:

Following example illustrates how extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces). This example is Generic method to return the largest of three Comparable objects:

public class MaximumTest

{

// determines the largest of three Comparable objects

public static <T extends Comparable<T>> T maximum(T x, T y, T z)

{

T max = x; // assume x is initially the largest

if ( y.compareTo( max ) > 0 ){

max = y; // y is the largest so far

}

if ( z.compareTo( max ) > 0 ){

max = z; // z is the largest now

}

return max; // returns the largest object

}

public static void main( String args[] )

{

System.out.printf( "Max of %d, %d and %d is %d\n\n",

3, 4, 5, maximum( 3, 4, 5 ) );

System.out.printf( "Maxm of %.1f,%.1f and %.1f is %.1f\n\n",

6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );

System.out.printf( "Max of %s, %s and %s is %s\n","pear",

"apple", "orange", maximum( "pear", "apple", "orange" ) );

}

}

This would produce the following result:

maximum of 3, 4 and 5 is 5

maximum of 6.6, 8.8 and 7.7 is 8.8

maximum of pear, apple and orange is pear

Generic Classes:

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Example:

Following example illustrates how we can define a generic class:

public class Box<T> {

private T t;

public void add(T t) {

this.t = t;

}

public T get() {

return t;

}

public static void main(String[] args) {

Box<Integer> integerBox = new Box<Integer>();

Box<String> stringBox = new Box<String>();

integerBox.add(new Integer(10));

stringBox.add(new String("Hello World"));

System.out.printf("Integer Value :%d\n\n", integerBox.get());

System.out.printf("String Value :%s\n", stringBox.get());

}

}

This would produce the following result:

Integer Value :10

String Value :Hello World

# Java - Serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out:

public final void writeObject(Object x) throws IOException

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object:

public final Object readObject() throws IOException, ClassNotFoundException

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

To demonstrate how serialization works in Java, I am going to use the Employee class that we discussed early on in the book. Suppose that we have the following Employee class, which implements the Serializable interface:

public class Employee implements java.io.Serializable

{

public String name;

public String address;

public transient int SSN;

public int number;

public void mailCheck()

{

System.out.println("Mailing a check to " + name + " " + address);

}

}

Notice that for a class to be serialized successfully, two conditions must be met:

* The class must implement the java.io.Serializable interface.
* All of the fields in the class must be serializable. If a field is not serializable, it must be marked **transient**.

If you are curious to know if a Java Standard Class is serializable or not, check the documentation for the class. The test is simple: If the class implements java.io.Serializable, then it is serializable; otherwise, it's not.

Serializing an Object:

The ObjectOutputStream class is used to serialize an Object. The following SerializeDemo program instantiates an Employee object and serializes it to a file.

When the program is done executing, a file named employee.ser is created. The program does not generate any output, but study the code and try to determine what the program is doing.

**Note:** When serializing an object to a file, the standard convention in Java is to give the file a **.ser**extension.

import java.io.\*;

public class SerializeDemo

{

public static void main(String [] args)

{

Employee e = new Employee();

e.name = "Reyan Ali";

e.address = "Phokka Kuan, Ambehta Peer";

e.SSN = 11122333;

e.number = 101;

try

{

FileOutputStream fileOut =

new FileOutputStream("/tmp/employee.ser");

ObjectOutputStream out = new ObjectOutputStream(fileOut);

out.writeObject(e);

out.close();

fileOut.close();

System.out.printf("Serialized data is saved in /tmp/employee.ser");

}catch(IOException i)

{

i.printStackTrace();

}

}

}

Deserializing an Object:

The following DeserializeDemo program deserializes the Employee object created in the SerializeDemo program. Study the program and try to determine its output:

import java.io.\*;

public class DeserializeDemo

{

public static void main(String [] args)

{

Employee e = null;

try

{

FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");

ObjectInputStream in = new ObjectInputStream(fileIn);

e = (Employee) in.readObject();

in.close();

fileIn.close();

}catch(IOException i)

{

i.printStackTrace();

return;

}catch(ClassNotFoundException c)

{

System.out.println("Employee class not found");

c.printStackTrace();

return;

}

System.out.println("Deserialized Employee...");

System.out.println("Name: " + e.name);

System.out.println("Address: " + e.address);

System.out.println("SSN: " + e.SSN);

System.out.println("Number: " + e.number);

}

}

This would produce the following result:

Deserialized Employee...

Name: Reyan Ali

Address:Phokka Kuan, Ambehta Peer

SSN: 0

Number:101

Here are following important points to be noted:

* The try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject() method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.
* Notice that the return value of readObject() is cast to an Employee reference.
* The value of the SSN field was 11122333 when the object was serialized, but because the field is transient, this value was not sent to the output stream. The SSN field of the deserialized Employee object is 0.

Wrapper class in Java

**Wrapper class in java** provides the mechanism *to convert primitive into object and object into primitive*.

Since J2SE 5.0, **autoboxing** and **unboxing** feature converts primitive into object and object into primitive automatically. The automatic conversion of primitive into object is known and autoboxing and vice-versa unboxing.

One of the eight classes of *java.lang* package are known as wrapper class in java. The list of eight wrapper classes are given below:

|  |  |
| --- | --- |
| **Primitive Type** | **Wrapper class** |
| boolean | Boolean |
| char | Character |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |

Wrapper class Example: Primitive to Wrapper

1. **public** **class** WrapperExample1{
2. **public** **static** **void** main(String args[]){
3. //Converting int into Integer
4. **int** a=20;
5. Integer i=Integer.valueOf(a);//converting int into Integer
6. Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
8. System.out.println(a+" "+i+" "+j);
9. }}

Output:

20 20 20

Wrapper class Example: Wrapper to Primitive

1. **public** **class** WrapperExample2{
2. **public** **static** **void** main(String args[]){
3. //Converting Integer to int
4. Integer a=**new** Integer(3);
5. **int** i=a.intValue();//converting Integer to int
6. **int** j=a;//unboxing, now compiler will write a.intValue() internally
8. System.out.println(a+" "+i+" "+j);
9. }}

Output: 3 3 3