



## CHAPTER 17

### *java.util and Subpackages*

This chapter documents the `java.util` package and each of its subpackages. Those packages are:

`java.util`

This package defines many commonly used utility classes, the most important of which are the various `Collection`, `Set`, `List`, and `Map` implementations.

`java.util.jar`

This package defines classes for reading and writing JAR (Java ARchive) files, which are based on the classes of the `java.util.zip` package.

`java.util.logging`

This package defines a powerful and flexible logging API for Java applications.

`java.util.prefs`

This package allows applications to set and query persistent values for user-specific preferences or system-wide configuration parameters.

`java.util.regex`

This package defines an API for textual pattern matching using regular expressions.

`java.util.zip`

This package defines classes for reading and writing ZIP files and for compressing and uncompressing data using the “gzip” format.

---

#### **Package `java.util`**

**Java 1.0**

The `java.util` package defines a number of useful classes, primarily collections classes that are useful for working with groups of objects. This package should not be considered merely a utility package that is separate from the rest of the language; it is an integral and frequently used part of the Java platform.

The most important classes in `java.util` are the collections classes. Prior to Java 1.2, these were `Vector`, a growable list of objects, and `Hashtable`, a mapping between arbitrary key

### *Package java.util*

and value objects. Java 1.2 adds an entire collections framework consisting of the `Collection`, `Map`, `Set`, `List`, `SortedMap`, and `SortedSet` interfaces and the classes that implement them. Other important classes and interfaces of the collections framework are `Comparator`, `Collections`, `Arrays`, `Iterator`, and `ListIterator`. Java 1.4 extends the Collections framework with the addition of new `Map` and `Set` implementations, and a new `RandomAccess` marker interface used by `List` implementations. `BitSet` is a related class that is not actually part of the Collections framework (and is not even a set). It provides a very compact representation of an arbitrary-size array or list of `boolean` values or bits. Its API was substantially enhanced in Java 1.4.

The other classes of the package are also quite useful. `Date`, `Calendar`, and `TimeZone` work with dates and times. `Currency` represents a national currency. `Locale` represents the language and related text formatting conventions of a country, region, or culture. `ResourceBundle` and its subclasses represent a bundle of localized resources that are read in by an internationalized program at runtime. `Random` generates and returns pseudo-random numbers in a variety of forms. `StringTokenizer` is a simple but surprisingly useful parser that breaks a string into tokens. In Java 1.3 and later, `Timer` and `TimerTask` provide a powerful API for scheduling code to be run by a background thread, once or repetitively, at a specified time in the future.

#### *Interfaces:*

```
public interface Comparator;  
public interface Enumeration;  
public interface Iterator;  
public interface ListIterator extends Iterator;  
public static interface Map.Entry;  
public interface Observer;  
public interface RandomAccess;
```

#### *Collections:*

```
public abstract class AbstractCollection implements Collection;  
    public abstract class AbstractList extends AbstractCollection implements List;  
        public abstract class AbstractSequentialList extends AbstractList;  
            public class LinkedList extends AbstractSequentialList  
                implements Cloneable, List, Serializable;  
        public class ArrayList extends AbstractList  
            implements Cloneable, List, RandomAccess, Serializable;  
        public class Vector extends AbstractList  
            implements Cloneable, List, RandomAccess, Serializable;  
        public class Stack extends Vector;  
    public abstract class AbstractSet extends AbstractCollection implements Set;  
        public class HashSet extends AbstractSet implements Cloneable, Serializable, Set;  
        public class LinkedHashSet extends HashSet implements Cloneable, Serializable, Set;  
        public class TreeSet extends AbstractSet implements Cloneable, Serializable, SortedSet;  
    public abstract class AbstractMap implements Map;  
        public class HashMap extends AbstractMap implements Cloneable, Map, Serializable;  
        public class LinkedHashMap extends HashMap;  
        public class IdentityHashMap extends AbstractMap implements Cloneable, Map, Serializable;  
        public class TreeMap extends AbstractMap implements Cloneable, Serializable, SortedMap;  
        public class WeakHashMap extends AbstractMap implements Map;  
    public interface Collection;  
    public class Hashtable extends Dictionary implements Cloneable, Map, Serializable;  
    public class Properties extends Hashtable;
```

```

public interface List extends Collection;
public interface Map;
public interface Set extends Collection;
public interface SortedMap extends Map;
public interface SortedSet extends Set;

```

*Events:*

```

public class EventObject implements Serializable;

```

*Event Listeners:*

```

public interface EventListener;

```

*Other Classes:*

```

public class Arrays;
public class BitSet implements Cloneable, Serializable;
public abstract class Calendar implements Cloneable, Serializable;
    public class GregorianCalendar extends Calendar;
public class Collections;
public final class Currency implements Serializable;
public class Date implements Cloneable, Comparable, Serializable;
public abstract class Dictionary;
public abstract class EventListenerProxy implements EventListener;
public final class Locale implements Cloneable, Serializable;
public class Observable;
public final class PropertyPermission extends java.security.BasicPermission;
public class Random implements Serializable;
public abstract class ResourceBundle;
    public abstract class ListResourceBundle extends ResourceBundle;
    public class PropertyResourceBundle extends ResourceBundle;
public class StringTokenizer implements Enumeration;
public class Timer;
public abstract class TimerTask implements Runnable;
public abstract class TimeZone implements Cloneable, Serializable;
    public class SimpleTimeZone extends TimeZone;

```

*Exceptions:*

```

public class ConcurrentModificationException extends RuntimeException;
public class EmptyStackException extends RuntimeException;
public class MissingResourceException extends RuntimeException;
public class NoSuchElementException extends RuntimeException;
public class TooManyListenersException extends Exception;

```

**AbstractCollection****Java 1.2**

java.util


*collection*

This abstract class is a partial implementation of `Collection` that makes it easy to define custom `Collection` implementations. To create an unmodifiable collection, simply override `size()` and `iterator()`. The `Iterator` object returned by `iterator()` has to support only the `hasNext()` and `next()` methods. To define a modifiable collection, you must additionally override the `add()` method of `AbstractCollection` and make sure the `Iterator` returned by `iterator()` supports the `remove()` method. Some subclasses may choose to override other methods to tune performance. In addition, it is conventional that all subclasses provide

## AbstractCollection

two constructors: one that takes no arguments and one that accepts a `Collection` argument that specifies the initial contents of the collection.

Note that if you subclass `AbstractCollection` directly, you are implementing a *bag*—an unordered collection that allows duplicate elements. If your `add()` method rejects duplicate elements, you should subclass `AbstractSet` instead. See also `AbstractList`.



```
classDiagram
    Object --> AbstractCollection
    AbstractCollection --> Collection
    class AbstractCollection {
        +protected AbstractCollection()
        +add(Object o) boolean
        +addAll(Collection c) boolean
        +clear() void
        +contains(Object o) boolean
        +containsAll(Collection c) boolean
        +isEmpty() boolean
        +iterator() Iterator
        +remove(Object o) boolean
        +removeAll(Collection c) boolean
        +retainAll(Collection c) boolean
        +size() int
        +toArray() Object[]
        +toArray(Object[] a) Object[]
        +toString() String
    }
```

```
public abstract class AbstractCollection implements Collection {
// Protected Constructors
    protected AbstractCollection();
// Methods Implementing Collection
    public boolean add(Object o);
    public boolean addAll(Collection c);
    public void clear();
    public boolean contains(Object o);
    public boolean containsAll(Collection c);
    public boolean isEmpty();
    public abstract Iterator iterator();
    public boolean remove(Object o);
    public boolean removeAll(Collection c);
    public boolean retainAll(Collection c);
    public abstract int size();
    public Object[] toArray();
    public Object[] toArray(Object[] a);
// Public Methods Overriding Object
    public String toString();
}
```

*Subclasses:* `AbstractList`, `AbstractSet`

## AbstractList

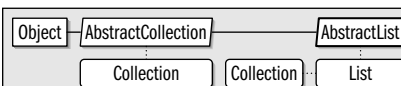
Java 1.2

`java.util`

*collection*

This abstract class is a partial implementation of the `List` interface that makes it easy to define custom `List` implementations based on random-access list elements (such as objects stored in an array). If you want to base a `List` implementation on a sequential-access data model (such as a linked list), subclass `AbstractSequentialList` instead.

To create an unmodifiable `List`, simply subclass `AbstractList` and override the (inherited) `size()` and `get()` methods. To create a modifiable list, you must also override `set()` and, optionally, `add()` and `remove()`. These three methods are optional, so unless you override them, they simply throw an `UnsupportedOperationException`. All other methods of the `List` interface are implemented in terms of `size()`, `get()`, `set()`, `add()`, and `remove()`. In some cases, you may want to override these other methods to improve performance. By convention, all `List` implementations should define two constructors: one that accepts no arguments and another that accepts a `Collection` of initial elements for the list.



```
classDiagram
    Object --> AbstractCollection
    AbstractCollection --> Collection
    AbstractList --|> AbstractCollection
    AbstractList --|> List
    Collection --> List
```

```
public abstract class AbstractList extends AbstractCollection implements java.util.List {
// Protected Constructors
    protected AbstractList();
// Methods Implementing List
    public boolean add(Object o);
}
```

```

    public void add(int index, Object element);
    public boolean addAll(int index, Collection c);
    public void clear();
    public boolean equals(Object o);
    public abstract Object get(int index);
    public int hashCode();
    public int indexOf(Object o);
    public Iterator iterator();
    public int lastIndexOf(Object o);
    public ListIterator listIterator();
    public ListIterator listIterator(int index);
    public Object remove(int index);
    public Object set(int index, Object element);
    public java.util.List subList(int fromIndex, int toIndex);
    // Protected Instance Methods
    protected void removeRange(int fromIndex, int toIndex);
    // Protected Instance Fields
    protected transient int modCount;
}

```

*Subclasses:* AbstractSequentialList, ArrayList, Vector

## AbstractMap

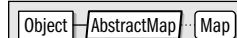
Java 1.2

java.util

collection

This abstract class is a partial implementation of the **Map** interface that makes it easy to define simple custom **Map** implementations. To define an unmodifiable map, subclass **AbstractMap** and override the **entrySet()** method so that it returns a set of **Map.Entry** objects. (Note that you must also implement **Map.Entry**, of course.) The returned set should not support **add()** or **remove()**, and its iterator should not support **remove()**. In order to define a modifiable **Map**, you must additionally override the **put()** method and provide support for the **remove()** method of the iterator returned by **entrySet().iterator()**. In addition, it is conventional that all **Map** implementations define two constructors: one that accepts no arguments and another that accepts a **Map** of initial mappings.

**AbstractMap** defines all **Map** methods in terms of its **entrySet()** and **put()** methods and the **remove()** method of the entry set iterator. Note, however, that the implementation is based on a linear search of the **Set** returned by **entrySet()** and is not efficient when the **Map** contains more than a handful of entries. Some subclasses may want to override additional **AbstractMap** methods to improve performance. **HashMap** and **TreeMap** use different algorithms and are substantially more efficient.



```

public abstract class AbstractMap implements Map {
    // Protected Constructors
    protected AbstractMap();
    // Methods Implementing Map
    public void clear();
    public boolean containsKey(Object key);
    public boolean containsValue(Object value);
    public abstract Set entrySet();
    public boolean equals(Object o);
    public Object get(Object key);
    public int hashCode();
    public boolean isEmpty();
}

```

## AbstractMap

```
public Set keySet();
public Object put(Object key, Object value);
public void putAll(Map t);
public Object remove(Object key);
public int size();
public Collection values();
// Public Methods Overriding Object
public String toString();
// Protected Methods Overriding Object
1.4 protected Object clone() throws CloneNotSupportedException;
}
```

*Subclasses:* HashMap, IdentityHashMap, TreeMap, WeakHashMap

## AbstractSequentialList

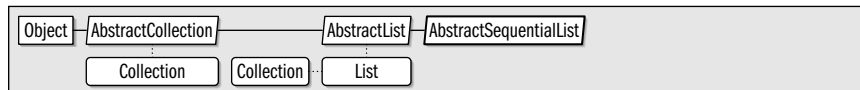
Java 1.2

java.util

collection

This abstract class is a partial implementation of the List interface that makes it easy to define List implementations based on a sequential-access data model, as is the case with the LinkedList subclass. To implement a List based on an array or other random-access model, subclass AbstractList instead.

To implement an unmodifiable list, subclass this class and override the size() and listIterator() methods. listIterator() must return a ListIterator that defines the hasNext(), hasPrevious(), next(), previous(), and index() methods. If you want to allow the list to be modified, the ListIterator should also support the set() method and, optionally, the add() and remove() methods. AbstractSequentialList implements all other List methods in terms of these methods. Some subclasses may want to override additional methods to improve performance. In addition, it is conventional that all List implementations define two constructors: one that accepts no arguments and another that accepts a Collection of initial elements for the list.



```
public abstract class AbstractSequentialList extends AbstractList {
// Protected Constructors
protected AbstractSequentialList();
// Public Methods Overriding AbstractList
public void add(int index, Object element);
public boolean addAll(int index, Collection c);
public Object get(int index);
public Iterator iterator();
public abstract ListIterator listIterator(int index);
public Object remove(int index);
public Object set(int index, Object element);
}
```

*Subclasses:* LinkedList

## AbstractSet

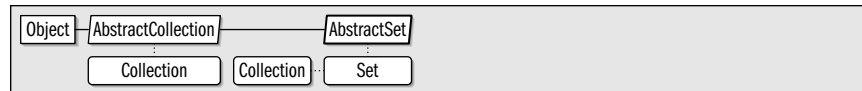
Java 1.2

java.util

collection

This abstract class is a partial implementation of the Set interface that makes it easy to create custom Set implementations. Since Set defines the same methods as Collection, you can subclass AbstractSet exactly as you would subclass AbstractCollection. See

AbstractCollection for details. Note, however, that when subclassing AbstractSet, you should be sure that your add() method and your constructors do not allow duplicate elements to be added to the set. See also AbstractList.



```

public abstract class AbstractSet extends AbstractCollection implements Set {
    // Protected Constructors
    protected AbstractSet();
    // Methods Implementing Set
    public boolean equals(Object o);
    public int hashCode();
    1.3 public boolean removeAll(Collection c);
}
  
```

*Subclasses:* HashSet, TreeSet

## ArrayList

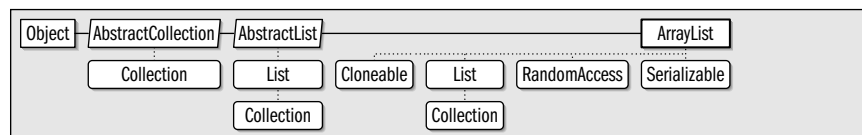
Java 1.2

java.util

cloneable serializable collection

This class is a List implementation based on an array (that is recreated as necessary as the list grows or shrinks). ArrayList implements all optional List and Collection methods and allows list elements of any type (including null). Because ArrayList is based on an array, the get() and set() methods are very efficient. (This is not the case for the LinkedList implementation, for example.) ArrayList is a general-purpose implementation of List and is quite commonly used. ArrayList is very much like the Vector class, except that its methods are not synchronized. If you are using an ArrayList in a multithreaded environment, you should explicitly synchronize any modifications to the list, or wrap the list with Collections.synchronizedList(). See List and Collection for details on the methods of ArrayList. See also LinkedList.

An ArrayList has a *capacity*, which is the number of elements in the internal array that contains the elements of the list. When the number of elements exceeds the capacity, a new array, with a larger capacity, must be created. In addition to the List and Collection methods, ArrayList defines a couple of methods that help you manage this capacity. If you know in advance how many elements an ArrayList will contain, you can call ensureCapacity(), which can increase efficiency by avoiding incremental reallocation of the internal array. You can also pass an initial capacity value to the ArrayList() constructor. Finally, if an ArrayList has reached its final size and will not change in the future, you can call trimToSize() to reallocate the internal array with a capacity that matches the list size exactly. When the ArrayList will have a long lifetime, this can be a useful technique to reduce memory usage.



```

public class ArrayList extends AbstractList implements Cloneable, java.util.List, RandomAccess, Serializable {
    // Public Constructors
    public ArrayList();
    public ArrayList(int initialCapacity);
    public ArrayList(Collection c);
    // Public Instance Methods
    public void ensureCapacity(int minCapacity);
}
  
```

## *ArrayList*

```
public void trimToSize();  
// Methods Implementing List  
public boolean add(Object o);  
public void add(int index, Object element);  
public boolean addAll(Collection c);  
public boolean addAll(int index, Collection c);  
public void clear();  
public boolean contains(Object elem);  
public Object get(int index);  
public int indexOf(Object elem);  
public boolean isEmpty(); default:true  
public int lastIndexOf(Object elem);  
public Object remove(int index);  
public Object set(int index, Object element);  
public int size();  
public Object[] toArray();  
public Object[] toArray(Object[] a);  
// Protected Methods Overriding AbstractList  
protected void removeRange(int fromIndex, int toIndex);  
// Public Methods Overriding Object  
public Object clone();  
}
```

**Returned By:** Collections.list()

**Type Of:** java.awt.dnd.DragGestureRecognizer.events,  
java.beans.beancontext.BeanContextServicesSupport.bcsListeners,  
java.beans.beancontext.BeanContextSupport.bcmListeners

## **Arrays**

**Java 1.2**

### **java.util**

This class defines static methods for sorting, searching, and performing other useful operations on arrays. It also defines the `asList()` method, which returns a `List` wrapper around a specified array of objects. Any changes made to the `List` are also made to the underlying array. This is a powerful method that allows any array of objects to be manipulated in any of the ways a `List` can be manipulated. It provides a link between arrays and the Java collections framework.

The various `sort()` methods sort an array (or a specified portion of an array) in place. Variants of the method are defined for arrays of each primitive type and for arrays of `Object`. For arrays of primitive types, the sorting is done according to the natural ordering of the type. For arrays of objects, the sorting is done according to the specified `Comparator`, or, if the array contains only `java.lang.Comparable` objects, according to the ordering defined by that interface. When sorting an array of objects, a stable sorting algorithm is used so that the relative ordering of equal objects is not disturbed. (This allows repeated sorts to order objects by key and subkey, for example.)

The `binarySearch()` methods perform an efficient search (in logarithmic time) of a sorted array for a specified value. If a match is found in the array, `binarySearch()` returns the index of the match. If no match is found, the method returns a negative number. For a negative return value `r`, the index `-(r+1)` specifies the array index at which the specified value can be inserted to maintain the sorted order of the array. When the array to be searched is an array of objects, the elements of the array must all implement `java.lang.Comparable`, or you must provide a `Comparator` object to compare them.

The `equals()` methods test whether two arrays are equal. Two arrays of primitive type are equal if they contain the same number of elements and if corresponding pairs of



elements are equal according to the `==` operator. Two arrays of objects are equal if they contain the same number of elements and if corresponding pairs of elements are equal according to the `equals()` method defined by those objects. The `fill()` methods fill an array or a specified range of an array with the specified value.

```
public class Arrays {
// No Constructor
// Public Class Methods
    public static java.util.List asList(Object[] a);
    public static int binarySearch(double[] a, double key);
    public static int binarySearch(byte[] a, byte key);
    public static int binarySearch(Object[] a, Object key);
    public static int binarySearch(float[] a, float key);
    public static int binarySearch(int[] a, int key);
    public static int binarySearch(long[] a, long key);
    public static int binarySearch(char[] a, char key);
    public static int binarySearch(short[] a, short key);
    public static int binarySearch(Object[] a, Object key, Comparator c);
    public static boolean equals(double[] a, double[] a2);
    public static boolean equals(boolean[] a, boolean[] a2);
    public static boolean equals(Object[] a, Object[] a2);
    public static boolean equals(float[] a, float[] a2);
    public static boolean equals(byte[] a, byte[] a2);
    public static boolean equals(int[] a, int[] a2);
    public static boolean equals(long[] a, long[] a2);
    public static boolean equals(char[] a, char[] a2);
    public static boolean equals(short[] a, short[] a2);
    public static void fill(short[] a, short val);
    public static void fill(char[] a, char val);
    public static void fill(long[] a, long val);
    public static void fill(int[] a, int val);
    public static void fill(byte[] a, byte val);
    public static void fill(float[] a, float val);
    public static void fill(Object[] a, Object val);
    public static void fill(boolean[] a, boolean val);
    public static void fill(double[] a, double val);
    public static void fill(short[] a, int fromIndex, int toIndex, short val);
    public static void fill(char[] a, int fromIndex, int toIndex, char val);
    public static void fill(long[] a, int fromIndex, int toIndex, long val);
    public static void fill(int[] a, int fromIndex, int toIndex, int val);
    public static void fill(byte[] a, int fromIndex, int toIndex, byte val);
    public static void fill(float[] a, int fromIndex, int toIndex, float val);
    public static void fill(Object[] a, int fromIndex, int toIndex, Object val);
    public static void fill(boolean[] a, int fromIndex, int toIndex, boolean val);
    public static void fill(double[] a, int fromIndex, int toIndex, double val);
    public static void sort(Object[] a);
    public static void sort(short[] a);
    public static void sort(char[] a);
    public static void sort(long[] a);
    public static void sort(byte[] a);
    public static void sort(float[] a);
    public static void sort(int[] a);
    public static void sort(double[] a);
    public static void sort(Object[] a, Comparator c);
    public static void sort(long[] a, int fromIndex, int toIndex);
    public static void sort(byte[] a, int fromIndex, int toIndex);
    public static void sort(char[] a, int fromIndex, int toIndex);
```

## Arrays

```
public static void sort(float[] a, int fromIndex, int toIndex);
public static void sort(int[] a, int fromIndex, int toIndex);
public static void sort(double[] a, int fromIndex, int toIndex);
public static void sort(Object[] a, int fromIndex, int toIndex);
public static void sort(short[] a, int fromIndex, int toIndex);
public static void sort(Object[] a, int fromIndex, int toIndex, Comparator c);
}
```

## BitSet

Java 1.0

java.util

*cloneable serializable*

This class implements an array or list of **boolean** values and stores them using a compact representation that requires only about 1 bit per value stored. It implements methods for setting, querying, and flipping the values stored at any given position within the list; for counting the number of **true** values stored in the list; and for finding the next **true** or **false** value in the list. It also defines a number of methods that perform bitwise boolean operations on two **BitSet** objects. Despite its name, **BitSet** does not implement the **Set** interface, nor does it have the behavior associated with a set; it is a list or vector for **boolean** values but is not related to the **List** interface or **Vector** class. This class was introduced in Java 1.0 but was substantially enhanced in Java 1.4; note that many of the following methods are available only in Java 1.4 and later.

Create a **BitSet** with the **BitSet()** constructor. You may optionally specify a size (the number of bits) for the **BitSet**, but this merely provides an optimization since a **BitSet** will grow as needed to accommodate any number of **boolean** values. **BitSet** does not define a precise notion of the size of a “set”. The **size()** method returns the number of **boolean** values that can be stored before more internal storage needs to be allocated. The **length()** method returns one more than the highest index of a set bit (i.e., a **true** value). This means that a **BitSet** that contains all **false** values will have a **length()** of 0. If your code needs to remember the index of the highest value stored in a **BitSet**, regardless of whether that value was **true** or **false**, then you should maintain that length information separately from the **BitSet**.

Set values in a **BitSet** with the **set()** method. There are four versions of this method. Two set the value at a specific index, and two set values for a range of indexes. Two of the **set()** methods do not take a value argument to set; they “set” the specified bit or range of bites, which means they store the value **true**. The other two methods take a **boolean** argument, allowing you to set the specified value or range of values to **true** (a set bit) or **false** (a clear bit). There are also two **clear()** methods that “clear” (or set to **false**) the value at the specified index or range of indexes. The **flip()** methods flip, or toggle (change **true** to **false** and **false** to **true**), the value or values at the specified index or range. The **set()**, **clear()**, and **flip()** methods, as well as all other **BitSet** methods that operate on a range of values, specify the range with two index values. They define the range as the values starting from, and including, the value stored at the first specified index up to, *but not including*, the value stored at the second specified index. (A number of methods of **String** and related classes follow the same convention for specifying a range of characters.)

To test the value stored at a specified location, use **get()**, which returns **true** if the specified bit is set, or **false** if it is not. There is also a **get()** method that specifies a range of bits and returns their state in the form of a **BitSet**; this **get()** method is analogous to the **substring()** method of a **String**. Because a **BitSet** does not define a maximum index, it is legal to pass any nonnegative value to **get()**. If the index you specify is greater than or equal to the value returned by **length()**, then the returned value will always be **false**.

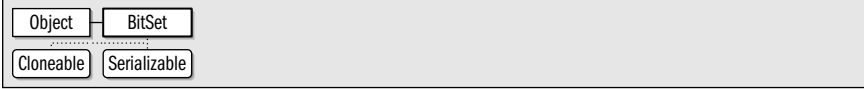
`cardinality()` returns the number of `true` values (or of set bits) stored in a `BitSet`. `isEmpty()` returns `true` if a `BitSet` has no `true` values stored in it (in this case, both `length()` and `cardinality()` return 0). `nextSetBit()` returns the first index at or after the specified index at which a `true` value is stored (or at which the bit is set). You can use this method in a loop to iterate through the indexes of `true` values. `nextClearBit()` is similar but searches the `BitSet` for `false` values (clear bits) instead. The `intersects()` method returns `true` if the target `BitSet` and the argument `BitSet` intersect, i.e., if there is at least one index at which both `BitSet` objects have a `true` value.

`BitSet` defines several methods that perform bitwise Boolean operations. These methods combine the `BitSet` on which they are invoked (called the “target” `BitSet`) with the `BitSet` passed as an argument and store the result in the target `BitSet`. If you want to perform a Boolean operation without altering the original `BitSet`, you should first make a copy of the original with the `clone()` method and invoke the method on the copy. The `and()` method performs a bitwise Boolean AND operation, much like the `&` does when applied to integer arguments. A value in the target `BitSet` will be `true` only if it was originally `true` and the value at the same index of the argument `BitSet` is also `true`. For all `false` values in the argument `BitSet`, `and()` sets the corresponding value in the target `BitSet` to `false`, leaving other values unchanged. The `andNot()` method combines a Boolean AND operation with a Boolean NOT operation on the argument `BitSet` (it does not alter the contents of that argument `BitSet`, however). The result is that for all `true` values in the argument `BitSet`, the corresponding values in the target `BitSet` are set to `false`.

The `or()` method performs a bitwise Boolean OR operation like the `|` operator: a value in the `BitSet` will be set to `true` if its original value was `true` or the corresponding value in the argument `BitSet` was `true`. For all `true` values in the argument `BitSet`, the `or()` method sets the corresponding value in the target `BitSet` to `true`, leaving the other values unchanged. The `xor()` method performs an “exclusive OR” operation: it sets a value in the target `BitSet` to `true` if it was originally `true` or if the corresponding value in the argument `BitSet` was `true`. If both values were `false`, or if both values were `true`, however, it sets the value to `false`.

Finally, the `toString()` method returns a `String` representation of a `BitSet` that consists of a list within curly braces of the indexes at which `true` values are stored.

The `BitSet` class is not thread-safe.



```

classDiagram
    class BitSet {
        <<abstract>>
        +Cloneable
        +Serializable
    }
    class Object
    class BitSet
    class Cloneable
    class Serializable
    Object --|> BitSet
    Cloneable ..|> BitSet
    Serializable ..|> BitSet
  
```

```

public class BitSet implements Cloneable, Serializable {
// Public Constructors
    public BitSet();
    public BitSet(int nbits);
// Public Instance Methods
    public void and(BitSet set);
1.2 public void andNot(BitSet set);
1.4 public int cardinality();
1.4 public void clear();
    public void clear(int bitIndex);
1.4 public void clear(int fromIndex, int toIndex);
1.4 public void flip(int bitIndex);
1.4 public void flip(int fromIndex, int toIndex);
    public boolean get(int bitIndex);
1.4 public BitSet get(int fromIndex, int toIndex);
1.4 public boolean intersects(BitSet set);
1.4 public boolean isEmpty();
}
  
```

*default:true*

## BitSet

```
1.2 public int length();
1.4 public int nextClearBit(int fromIndex);
1.4 public int nextSetBit(int fromIndex);
    public void or(BitSet set);
    public void set(int bitIndex);
1.4 public void set(int bitIndex, boolean value);
1.4 public void set(int fromIndex, int toIndex);
1.4 public void set(int fromIndex, int toIndex, boolean value);
    public int size();
    public void xor(BitSet set);
// Public Methods Overriding Object
    public Object clone();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}
```

*Passed To:* BitSet.{and(), andNot(), intersects(), or(), xor()},  
javax.swing.text.html.parser.DTD.defineElement()

*Returned By:* BitSet.get()

*Type Of:* javax.swing.text.html.parser.Element.{exclusions, inclusions}

## Calendar

Java 1.1

java.util

*cloneable serializable*

This abstract class defines methods that perform date and time arithmetic. It also includes methods that convert dates and times to and from the machine-usable millisecond format used by the **Date** class and units such as minutes, hours, days, weeks, months, and years that are more useful to humans. As an abstract class, **Calendar** cannot be directly instantiated. Instead, it provides static **getInstance()** methods that return instances of a **Calendar** subclass suitable for use in a specified or default locale with a specified or default time zone. See also **Date**, **DateFormat**, and **TimeZone**.

**Calendar** defines a number of useful constants. Some of these are values that represent days of the week and months of the year. Other constants, such as  **HOUR** and  **DAY\_OF\_WEEK**, represent various fields of date and time information. These field constants are passed to a number of **Calendar** methods, such as **get()** and **set()**, in order to indicate what particular date or time field is desired.

The **add()** method adds (or subtracts) values to a calendar field, incrementing the next larger field when the field being set rolls over. **roll()** does the same, without modifying anything but the specified field. **before()** and **after()** compare two **Calendar** objects. Many of the methods of the **Calendar** class are replacements for methods of **Date** that have been deprecated as of Java 1.1. While the **Calendar** class converts a time value to its various hour, day, month, and other fields, it is not intended to present those fields in a form suitable for display to the end user. That function is performed by the **java.text.DateFormat** class, which handles internationalization issues.

Object	Calendar
Cloneable	Serializable

```
public abstract class Calendar implements Cloneable, Serializable {
// Protected Constructors
    protected Calendar();
```

```

protected Calendar(TimeZone zone, Locale aLocale);
// Public Constants
public static final int AM;                                =0
public static final int AM_PM;                            =9
public static final int APRIL;                            =3
public static final int AUGUST;                           =7
public static final int DATE;                             =5
public static final int DAY_OF_MONTH;                     =5
public static final int DAY_OF_WEEK;                     =7
public static final int DAY_OF_WEEK_IN_MONTH;            =8
public static final int DAY_OF_YEAR;                     =6
public static final int DECEMBER;                        =11
public static final int DST_OFFSET;                      =16
public static final int ERA;                              =0
public static final int FEBRUARY;                        =1
public static final int FIELD_COUNT;                    =17
public static final int FRIDAY;                          =6
public static final int HOUR;                            =10
public static final int HOUR_OF_DAY;                    =11
public static final int JANUARY;                        =0
public static final int JULY;                            =6
public static final int JUNE;                            =5
public static final int MARCH;                           =2
public static final int MAY;                             =4
public static final int MILLISECOND;                   =14
public static final int MINUTE;                         =12
public static final int MONDAY;                         =2
public static final int MONTH;                          =2
public static final int NOVEMBER;                      =10
public static final int OCTOBER;                       =9
public static final int PM;                              =1
public static final int SATURDAY;                      =7
public static final int SECOND;                         =13
public static final int SEPTEMBER;                     =8
public static final int SUNDAY;                        =1
public static final int THURSDAY;                      =5
public static final int TUESDAY;                       =3
public static final int UNDECIMBER;                    =12
public static final int WEDNESDAY;                     =4
public static final int WEEK_OF_MONTH;                 =4
public static final int WEEK_OF_YEAR;                  =3
public static final int YEAR;                          =1
public static final int ZONE_OFFSET;                   =15
// Public Class Methods
public static Locale[] getAvailableLocales();            synchronized
public static Calendar getInstance();
public static Calendar getInstance(TimeZone zone);
public static Calendar getInstance(Locale aLocale);
public static Calendar getInstance(TimeZone zone, Locale aLocale);
// Property Accessor Methods (by property name)
public int getFirstDayOfWeek();
public void setFirstDayOfWeek(int value);
public boolean isLenient();
public void setLenient(boolean lenient);
public int getMinimalDaysInFirstWeek();

```

## Calendar

```
public void setMinimalDaysInFirstWeek(int value);
public final java.util.Date getTime();
public final void setTime(java.util.Date date);
public long getTimeInMillis();
public void setTimeInMillis(long millis);
public TimeZone getTimeZone();
public void setTimeZone(TimeZone value);
// Public Instance Methods
public abstract void add(int field, int amount);
public boolean after(Object when);
public boolean before(Object when);
public final void clear();
public final void clear(int field);
public int get(int field);
1.2 public int getActualMaximum(int field);
1.2 public int getActualMinimum(int field);
public abstract int getGreatestMinimum(int field);
public abstract int getLeastMaximum(int field);
public abstract int getMaximum(int field);
public abstract int getMinimum(int field);
public final boolean isSet(int field);
public abstract void roll(int field, boolean up);
1.2 public void roll(int field, int amount);
public void set(int field, int value);
public final void set(int year, int month, int date);
public final void set(int year, int month, int date, int hour, int minute);
public final void set(int year, int month, int date, int hour, int minute, int second);
// Public Methods Overriding Object
public Object clone();
public boolean equals(Object obj);
1.2 public int hashCode();
public String toString();
// Protected Instance Methods
protected void complete();
protected abstract void computeFields();
protected abstract void computeTime();
protected final int internalGet(int field);
// Protected Instance Fields
protected boolean areFieldsSet;
protected int[] fields;
protected boolean[] isSet;
protected boolean isTimeSet;
protected long time;
}
```

*Subclasses:* `GregorianCalendar`

*Passed To:* Too many methods to list.

*Returned By:* `java.text.DateFormat.getCalendar()`, `Calendar.getInstance()`

*Type Of:* `java.text.DateFormat.calendar`

**Collection**

Java 1.2

java.util

collection

This interface represents a group, or collection, of objects. The objects may or may not be ordered, and the collection may or may not contain duplicate objects. **Collection** is not often implemented directly. Instead, most collection classes implement one of the more specific subinterfaces: **Set**, an unordered collection that does not allow duplicates, or **List**, an ordered collection that does allow duplicates.

The **Collection** type provides a general way to refer to any set, list, or other collection of objects; it defines generic methods that work with any collection. **contains()** and **containsAll()** test whether the **Collection** contains a specified object or all the objects in a given collection. **isEmpty()** returns **true** if the **Collection** has no elements, or **false** otherwise. **size()** returns the number of elements in the **Collection**. **iterator()** returns an **Iterator** object that allows you to iterate through the objects in the collection. **toArray()** returns the objects in the **Collection** in a new array of type **Object**. Another version of **toArray()** takes an array as an argument and stores all elements of the **Collection** (which must all be compatible with the array) into that array. If the array is not big enough, the method allocates a new, larger array of the same type. If the array is too big, the method stores null into the first empty element of the array. This version of **toArray()** returns the array that was passed in or the new array, if one was allocated.

The previous methods all query or extract the contents of a collection. The **Collection** interface also defines methods for modifying the contents of the collection. **add()** and **addAll()** add an object or a collection of objects to a **Collection**. **remove()** and **removeAll()** remove an object or collection. **retainAll()** is a variant that removes all objects except those in a specified **Collection**. **clear()** removes all objects from the collection. All these modification methods except **clear()** return **true** if the collection was modified as a result of the call. An interface cannot specify constructors, but it is conventional that all implementations of **Collection** provide at least two standard constructors: one that takes no arguments and creates an empty collection, and a copy constructor that accepts a **Collection** object that specifies the initial contents of the new **Collection**.

Implementations of **Collection** and its subinterfaces are not required to support all operations defined by the **Collection** interface. All modification methods listed above are optional; an implementation (such as an immutable **Set** implementation) that does not support them simply throws **java.lang.UnsupportedOperationException** for these methods. Furthermore, implementations are free to impose restrictions on the types of objects that can be members of a collection. Some implementations might require elements to be of a particular type, for example, and others might not allow null as an element.

See also **Set**, **List**, **Map**, and **Collections**.

```
public interface Collection {
// Public Instance Methods
    public abstract boolean add(Object o);
    public abstract boolean addAll(Collection c);
    public abstract void clear();
    public abstract boolean contains(Object o);
    public abstract boolean containsAll(Collection c);
    public abstract boolean equals(Object o);
    public abstract int hashCode();
    public abstract boolean isEmpty();
    public abstract Iterator iterator();
    public abstract boolean remove(Object o);
    public abstract boolean removeAll(Collection c);
    public abstract boolean retainAll(Collection c);
```

## Collection

```
public abstract int size();
public abstract Object[] toArray();
public abstract Object[] toArray(Object[] a);
}
```

*Implementations:* java.beans.beancontext.BeanContext, AbstractCollection, java.util.List, Set

*Passed To:* Too many methods to list.

*Returned By:* Too many methods to list.

*Type Of:* java.beans.beancontext.BeanContextMembershipEvent.children

## Collections

Java 1.2

### java.util

This class defines static methods and constants that are useful for working with collections and maps. One of the most commonly used methods is `sort()`, which sorts a `List` in place (the list cannot be immutable, of course). The sorting algorithm is stable, which means that equal elements retain the same relative order. One version of `sort()` uses a specified `Comparator` to perform the sort; the other relies on the natural ordering of the list elements and requires all the elements to implement `java.lang.Comparable`. `reverseOrder()` returns a convenient predefined `Comparator` object that can order `Comparable` objects into the reverse of their natural ordering.

A related method is `binarySearch()`. It efficiently (in logarithmic time) searches a sorted `List` for a specified object and returns the index at which a matching object is found. If no match is found, it returns a negative number. For a negative return value  $r$ , the value  $-(r+1)$  specifies the index at which the specified object can be inserted into the list to maintain the sorted order of the list. As with `sort()`, `binarySearch()` can be passed a `Comparator` that defines the order of the sorted list. If no `Comparator` is specified, the list elements must all implement `Comparable`, and the list is assumed to be sorted according to the natural ordering defined by this interface.

See *Arrays* for methods that perform sorting and searching operations on arrays instead of collections.

The various methods with names beginning with `synchronized` return a thread-safe collection object wrapped around the specified collection. `Vector` and `Hashtable` are the only two collection objects that are thread-safe by default. Use these methods to obtain a `synchronized` wrapper object if you are using any other type of `Collection` or `Map` in a multithreaded environment where more than one thread can modify it.

The various methods whose names begin with `unmodifiable` function like `synchronized` methods. They return a `Collection` or `Map` object wrapped around the specified collection. The returned object is unmodifiable, however, so its `add()`, `remove()`, `set()`, `put()`, etc., methods all throw `java.lang.UnsupportedOperationException`.

In addition to the “synchronized” and “unmodifiable” methods, `Collections` defines a number of other methods that return special-purpose collections or maps. `singleton()` returns an unmodifiable set that contains only the specified object. `singletonList()` and `singletonMap()` return an immutable list and an immutable map, respectively, each of which contains only a single entry. The `Collections` class also defines related constants—`EMPTY_LIST`, `EMPTY_SET`, and `EMPTY_MAP`—which are immutable `List`, `Set`, and `Map` objects that contain no elements or mappings. `nCopies()` creates a new immutable `List` that contains a specified number of copies of a specified object. `list()` returns a `List` object that represents the elements of the specified `Enumeration` object. `enumeration()` does the reverse: it returns an `Enumeration` for a `Collection`, which is useful when working with code that uses the old `Enumeration` interface instead of the newer `Iterator` interface.



The `Collections` class also defines methods that mutate a collection. These methods throw an `UnsupportedOperationException` if the target collection does not allow mutation. `copy()` copies elements of a source list into a destination list. `fill()` replaces all elements of the specified list with the specified object. `swap()` swaps the elements at two specified indexes of a `List`. `replaceAll()` replaces all elements in a `List` that are equal to (using the `equals()` method) with another object and returns `true` if any replacements were done. `reverse()` reverses the order of the elements in a list. `rotate()` “rotates” a list, adding the specified number to the index of each element and wrapping elements from the end of the list back to the front of the list. (Specifying a negative rotation rotates the list in the other direction.) `shuffle()` randomizes the order of elements in a list, using either an internal source of randomness or the `Random` pseudo-random number generator that you provide.

Finally, `Collections` defines methods (in addition to the `binarySearch()` methods described earlier) that search the elements of a collection; `min()` and `max()` methods search an unordered `Collection` for the minimum and maximum elements, according to either a specified `Comparator` or to the natural order defined by the `Comparable` elements themselves. `indexOfSubList()` and `lastIndexOfSubList()` search a specified list forward or backward for a subsequence of elements that match (using `equals()`) the elements of a second specified list. They return the start index of any such matching sublist, or return `-1` if no match was found. These methods are like the `indexOf()` and `lastIndexOf()` methods of `String` and do not require the `List` to be sorted, as the `binarySearch()` methods do.

```
public class Collections {
    // No Constructor
    // Public Constants
    public static final java.util.List EMPTY_LIST;
    1.3 public static final Map EMPTY_MAP;
    public static final Set EMPTY_SET;
    // Public Class Methods
    public static int binarySearch(java.util.List list, Object key);
    public static int binarySearch(java.util.List list, Object key, Comparator c);
    public static void copy(java.util.List dest, java.util.List src);
    public static Enumeration enumeration(Collection c);
    public static void fill(java.util.List list, Object obj);
    1.4 public static int indexOfSubList(java.util.List source, java.util.List target);
    1.4 public static int lastIndexOfSubList(java.util.List source, java.util.List target);
    1.4 public static ArrayList list(Enumeration e);
    public static Object max(Collection coll);
    public static Object max(Collection coll, Comparator comp);
    public static Object min(Collection coll);
    public static Object min(Collection coll, Comparator comp);
    public static java.util.List nCopies(int n, Object o);
    1.4 public static boolean replaceAll(java.util.List list, Object oldVal, Object newVal);
    public static void reverse(java.util.List list);
    public static Comparator reverseOrder();
    1.4 public static void rotate(java.util.List list, int distance);
    public static void shuffle(java.util.List list);
    public static void shuffle(java.util.List list, Random rnd);
    public static Set singleton(Object o);
    1.3 public static java.util.List singletonList(Object o);
    1.3 public static Map singletonMap(Object key, Object value);
    public static void sort(java.util.List list);
    public static void sort(java.util.List list, Comparator c);
    1.4 public static void swap(java.util.List list, int i, int j);
    public static Collection synchronizedCollection(Collection c);
}
```

## Collections

```
public static java.util.List synchronizedList(java.util.List list);  
public static Map synchronizedMap(Map m);  
public static Set synchronizedSet(Set s);  
public static SortedMap synchronizedSortedMap(SortedMap m);  
public static SortedSet synchronizedSortedSet(SortedSet s);  
public static Collection unmodifiableCollection(Collection c);  
public static java.util.List unmodifiableList(java.util.List list);  
public static Map unmodifiableMap(Map m);  
public static Set unmodifiableSet(Set s);  
public static SortedMap unmodifiableSortedMap(SortedMap m);  
public static SortedSet unmodifiableSortedSet(SortedSet s);  
}
```

## Comparator

Java 1.2

java.util

This interface defines a `compare()` method that specifies a total ordering for a set of objects, allowing those objects to be sorted. The `Comparator` is used when the objects to be ordered do not have a natural ordering defined by the `Comparable` interface, or when you want to order them using something other than their natural ordering.

The `compare()` method is passed two objects. If the first argument is less than the second argument or should be placed before the second argument in a sorted list, `compare()` should return a negative integer. If the first argument is greater than the second argument or should be placed after the second argument in a sorted list, `compare()` should return a positive integer. If the two objects are equivalent or if their relative position in a sorted list does not matter, `compare()` should return 0. `Comparator` implementations may assume that both `Object` arguments are of appropriate types and cast them as desired. If either argument is not of the expected type, the `compare()` method throws a `ClassCastException`.

Note that the magnitude of the numbers returned by `compare()` does not matter, only whether they are less than, equal to, or greater than zero. In most cases, you should implement a `Comparator` so that `compare(o1,o2)` returns 0 if and only if `o1.equals(o2)` returns `true`. This is particularly important when using a `Comparator` to impose an ordering on a `TreeSet` or a `TreeMap`.

See `Collections` and `Arrays` for various methods that use `Comparator` objects for sorting and searching. See also the related `java.lang.Comparable` interface.

```
public interface Comparator {  
    // Public Instance Methods  
    public abstract int compare(Object o1, Object o2);  
    public abstract boolean equals(Object obj);  
}
```

**Implementations:** `java.text.Collator`

**Passed To:** `Arrays.binarySearch()`, `sort()`, `Collections.binarySearch()`, `max()`, `min()`, `sort()`, `TreeMap.TreeMap()`, `TreeSet.TreeSet()`, `javax.swing.SortingFocusTraversalPolicy.setComparator()`, `SortingFocusTraversalPolicy()`

**Returned By:** `Collections.reverseOrder()`, `SortedMap.comparator()`, `SortedSet.comparator()`, `TreeMap.comparator()`, `TreeSet.comparator()`, `javax.swing.SortingFocusTraversalPolicy.getComparator()`

**Type Of:** `String.CASE_INSENSITIVE_ORDER`

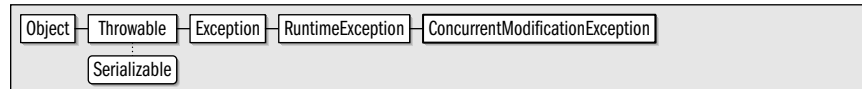
**ConcurrentModificationException**

Java 1.2

java.util

serializable unchecked

This exception signals that a modification has been made to a data structure at the same time some other operation is in progress and that, as a result, the correctness of the ongoing operation cannot be guaranteed. It is typically thrown by an `Iterator` or `ListIterator` object to stop an iteration if it detects that the underlying collection has been modified while the iteration is in progress.



```

public class ConcurrentModificationException extends RuntimeException {
    // Public Constructors
    public ConcurrentModificationException();
    public ConcurrentModificationException(String message);
}

```

**Currency**

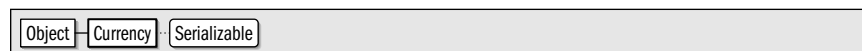
Java 1.4

java.util

serializable

Instances of this class represent a currency. Obtain a `Currency` object by passing a “currency code,” such as “USD” for U.S. dollars or “EUR” for Euros, to `getInstance()`. Once you have a `Currency` object, use `getSymbol()` to obtain the currency symbol (which is often different from the currency code) for the default locale or for a specified `Locale`. For example, the symbol for a USD would be “\$” in a U.S. locale but might be “US\$” in other locales. If no symbol is known, this method returns the currency code.

Use `getDefaultFractionDigits()` to determine how many fractional digits are conventionally used with the currency. This method returns 2 for the U.S. dollar and other currencies that are divided into hundredths but returns 3 for the Jordanian Dinar (JOD) and other currencies that are traditionally divided into thousandths. It returns 0 for the Japanese Yen (JPY) and other currencies that have a small unit value and are not usually divided into fractional parts at all. Currency codes are standardized by the ISO 4217 standard. For a complete list of currencies and currency codes, see the web site of the maintenance agency for this standard at [http://www.bsi-global.com/Technical+Information/Publications/\\_Publications/tig90.xalter](http://www.bsi-global.com/Technical+Information/Publications/_Publications/tig90.xalter).



```

public final class Currency implements Serializable {
    // No Constructor
    // Public Class Methods
    public static Currency getInstance(String currencyCode);
    public static Currency getInstance(Locale locale);
    // Public Instance Methods
    public String getCurrencyCode();
    public int getDefaultFractionDigits();
    public String getSymbol();
    public String getSymbol(Locale locale);
    // Public Methods Overriding Object
    public String toString();
}

```

*Passed To:* `java.text.DecimalFormat.setCurrency()`, `java.text.DecimalFormatSymbols.setCurrency()`, `java.text.NumberFormat.setCurrency()`

## Currency

**Returned By:** `java.text.DecimalFormat.getCurrency()`, `java.text.DecimalFormatSymbols.getCurrency()`, `java.text.NumberFormat.getCurrency()`, `Currency.getInstance()`

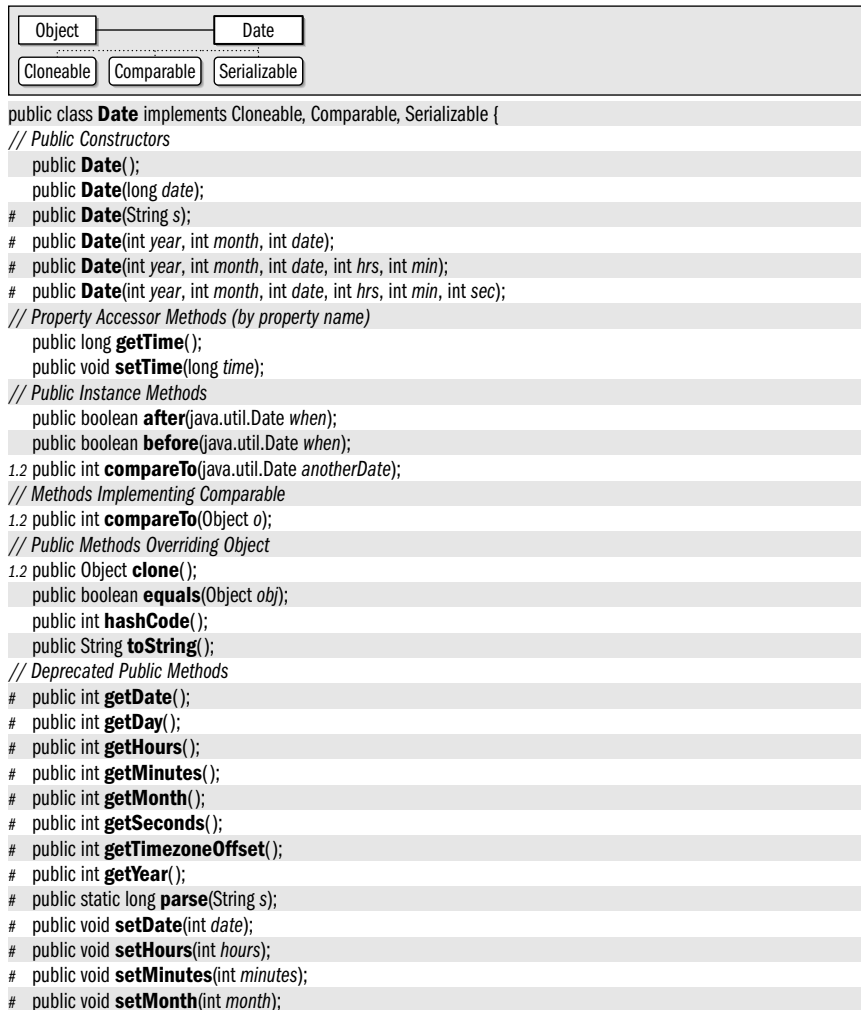
## Date

Java 1.0

`java.util`

*cloneable serializable comparable*

This class represents dates and times and lets you work with them in a system-independent way. You can create a `Date` by specifying the number of milliseconds from the epoch (midnight GMT, January 1st, 1970) or the year, month, date, and, optionally, the hour, minute, and second. Years are specified as the number of years since 1900. If you call the `Date` constructor with no arguments, the `Date` is initialized to the current time and date. The instance methods of the class allow you to get and set the various date and time fields, to compare dates and times, and to convert dates to and from string representations. As of Java 1.1, many of the date methods have been deprecated in favor of the methods of the `Calendar` class.



## EmptyStackException

```
# public void setSeconds(int seconds);
# public void setYear(int year);
# public String toGMTString();
# public String toLocaleString();
# public static long UTC(int year, int month, int date, int hrs, int min, int sec);
}
```

*Subclasses:* java.sql.Date, java.sql.Time, java.sql.Timestamp

*Passed To:* Too many methods to list.

*Returned By:* Too many methods to list.

## Dictionary

Java 1.0

java.util

This abstract class is the superclass of `Hashtable`. Other hashtable-like data structures might also extend this class. See `Hashtable` for more information. In Java 1.2, the `Map` interface replaces the functionality of this class.

```
public abstract class Dictionary {
    // Public Constructors
    public Dictionary();
    // Public Instance Methods
    public abstract Enumeration elements();
    public abstract Object get(Object key);
    public abstract boolean isEmpty();
    public abstract Enumeration keys();
    public abstract Object put(Object key, Object value);
    public abstract Object remove(Object key);
    public abstract int size();
}
```

*Subclasses:* `Hashtable`

*Passed To:* javax.swing.JSlider.setLabelTable(),  
javax.swing.text.AbstractDocument.setDocumentProperties()

*Returned By:* javax.swing.JSlider.getLabelTable(),  
javax.swing.text.AbstractDocument.getDocumentProperties()

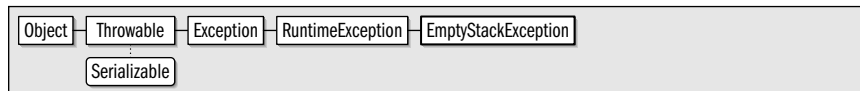
## EmptyStackException

Java 1.0

java.util

*serializable unchecked*

This exception signals that a `Stack` object is empty.



```
public class EmptyStackException extends RuntimeException {
    // Public Constructors
    public EmptyStackException();
}
```

*Thrown By:* java.awt.EventQueue.pop()

**Enumeration****Java 1.0****java.util**

This interface defines the methods necessary to enumerate, or iterate, through a set of values, such as the set of values contained in a hashtable or binary tree. It is particularly useful for data structures, like hashtables, for which elements cannot simply be looked up by index, as they can in arrays. An **Enumeration** is usually not instantiated directly, but instead is created by the object that is to have its values enumerated. A number of classes, such as **Vector** and **Hashtable**, have methods that return **Enumeration** objects. In Java 1.2, the new **Iterator** interface is preferred over **Enumeration**.

To use an **Enumeration** object, you use its two methods in a loop. **hasMoreElements()** returns **true** if there are more values to be enumerated and can determine whether a loop should continue. Within a loop, a call to **nextElement()** returns a value from the enumeration. An **Enumeration** makes no guarantees about the order in which the values are returned. The values in an **Enumeration** can be iterated through only once; there is no way to reset it to the beginning.

```
public interface Enumeration {
    // Public Instance Methods
    public abstract boolean hasMoreElements();
    public abstract Object nextElement();
}
```

**Implementations:** java.text.CharSet.Enumeration, StringTokenizer, javax.naming.NamingEnumeration

**Passed To:** java.io.SequenceInputStream.SequenceInputStream(), Collections.list(), javax.naming.CompositeName.CompositeName(), javax.naming.CompoundName.CompoundName(), javax.swing.JTree.removeDescendantToggledPaths(), javax.swing.text.AbstractDocument.AbstractElement.removeAttributes(), javax.swing.text.AbstractDocument.AttributeContext.removeAttributes(), javax.swing.text.MutableAttributeSet.removeAttributes(), javax.swing.text.SimpleAttributeSet.removeAttributes(), javax.swing.text.StyleContext.removeAttributes(), javax.swing.text.StyleContext.NamedStyle.removeAttributes(), javax.swing.text.html.StyleSheet.removeAttributes()

**Returned By:** Too many methods to list.

**Type Of:** javax.swing.tree.DefaultMutableTreeNode.EMPTY\_ENUMERATION

**EventListener****Java 1.1****java.util****event listener**

**EventListener** is a base interface for the event model that is used by AWT and Swing in Java 1.1 and later. This interface defines no methods or constants; it serves simply as a tag that identifies objects that act as event listeners. The event listener interfaces in the java.awt.event, java.beans, and javax.swing.event packages extend this interface.

```
public interface EventListener {
}
```

**Implementations:** Too many classes to list.

**Passed To:** java.awt.AWTEventMulticaster.{addInternal(), AWTEventMulticaster(), getListeners(), remove(), removeInternal(), save()}, EventListenerProxy.EventListenerProxy(), javax.swing.event.EventListenerList.{add(), remove()}

**Returned By:** Too many methods to list.

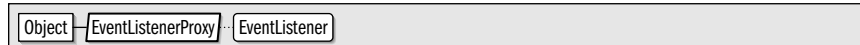
Type Of: java.awt.AWTEventMulticaster.{a, b}

## EventListenerProxy

Java 1.4

java.util

This abstract class serves as the superclass for event listener proxy objects. Subclasses of this class implement an event listener interface and serve as a wrapper around an event listener of that type, defining methods that provide additional information about the listener. See java.beans.PropertyChangeListenerProxy for an explanation of how event listener proxy objects are used.



```

public abstract class EventListenerProxy implements java.util.EventListener {
    // Public Constructors
    public EventListenerProxy(java.util.EventListener listener);
    // Public Instance Methods
    public java.util.EventListener getListener();
}
  
```

Subclasses: java.awt.event.AWTEventListenerProxy, java.beans.PropertyChangeListenerProxy, java.beans.VetoableChangeListenerProxy

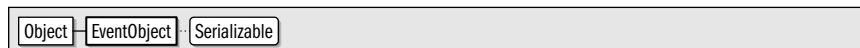
## EventObject

Java 1.1

java.util

serializable event

EventObject serves as the superclass for all event objects used by the event model introduced in Java 1.1 for AWT and JavaBeans and also used by Swing in Java 1.2. This class defines a generic type of event; it is extended by the more specific event classes in the java.awt, java.awt.event, java.beans, and javax.swing.event packages. The only common feature shared by all events is a source object, which is the object that, in some way, generated the event. The source object is passed to the EventObject() constructor and is returned by the getSource() method.



```

public class EventObject implements Serializable {
    // Public Constructors
    public EventObject(Object source);
    // Public Instance Methods
    public Object getSource();
    // Public Methods Overriding Object
    public String toString();
    // Protected Instance Fields
    protected transient Object source;
}
  
```

Subclasses: Too many classes to list.

Passed To: javax.swing.AbstractCellEditor.{isCellEditable(), shouldSelectCell()}, javax.swing.CellEditor.{isCellEditable(), shouldSelectCell()}, javax.swing.DefaultCellEditor.{isCellEditable(), shouldSelectCell()}, javax.swing.DefaultCellEditor.EditorDelegate.{isCellEditable(), shouldSelectCell(), startCellEditing()}, javax.swing.JTable.editCellAt(), javax.swing.tree.DefaultTreeCellEditor.{canEditImmediately(), isCellEditable(), shouldSelectCell(), shouldStartEditingTimer()}


**GregorianCalendar**

Java 1.1

java.util

cloneable serializable

This concrete subclass of `Calendar` implements the standard solar calendar with years numbered from the birth of Christ that is used in most locales throughout the world. You do not typically use this class directly, but instead obtain a `Calendar` object suitable for the default locale by calling `Calendar.getInstance()`. See `Calendar` for details on working with `Calendar` objects. There is a discontinuity in the Gregorian calendar that represents the historical switch from the Julian calendar to the Gregorian calendar. By default, `GregorianCalendar` assumes that this switch occurs on October 15, 1582. Most programs need not be concerned with the switch.



```

public class GregorianCalendar extends Calendar {
    // Public Constructors
    public GregorianCalendar();
    public GregorianCalendar(TimeZone zone);
    public GregorianCalendar(Locale aLocale);
    public GregorianCalendar(TimeZone zone, Locale aLocale);
    public GregorianCalendar(int year, int month, int date);
    public GregorianCalendar(int year, int month, int date, int hour, int minute);
    public GregorianCalendar(int year, int month, int date, int hour, int minute, int second);

    // Public Constants
    public static final int AD;                =1
    public static final int BC;                =0

    // Public Instance Methods
    public final java.util.Date getGregorianChange();
    public boolean isLeapYear(int year);
    public void setGregorianChange(java.util.Date date);

    // Public Methods Overriding Calendar
    public void add(int field, int amount);
    public boolean equals(Object obj);
    1.2 public int getActualMaximum(int field);
    1.2 public int getActualMinimum(int field);
    public int getGreatestMinimum(int field);
    public int getLeastMaximum(int field);
    public int getMaximum(int field);
    public int getMinimum(int field);
    public int hashCode();
    1.2 public void roll(int field, int amount);
    public void roll(int field, boolean up);

    // Protected Methods Overriding Calendar
    protected void computeFields();
    protected void computeTime();
}

```

**HashMap**

Java 1.2

java.util

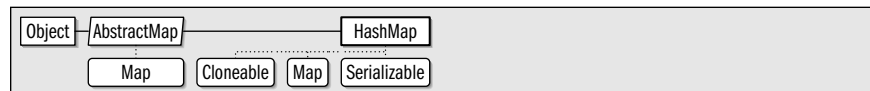
cloneable serializable collection

This class implements the `Map` interface using an internal hashtable. It supports all optional `Map` methods, allows key and value objects of any types, and allows null to be used as a key or a value. Because `HashMap` is based on a hashtable data structure, the `get()` and `put()` methods are very efficient. `HashMap` is much like the `Hashtable` class, except that the `HashMap` methods are not synchronized (and are therefore faster), and



HashMap allows null to be used as a key or a value. If you are working in a multi-threaded environment, or if compatibility with previous versions of Java is a concern, use Hashtable. Otherwise, use HashMap.

If you know in advance approximately how many mappings a HashMap will contain, you can improve efficiency by specifying *initialCapacity* when you call the `HashMap()` constructor. The *initialCapacity* argument times the *loadFactor* argument should be greater than the number of mappings the HashMap will contain. A good value for *loadFactor* is 0.75; this is also the default value. See Map for details on the methods of HashMap. See also TreeMap and HashSet.



```

public class HashMap extends AbstractMap implements Cloneable, Map, Serializable {
// Public Constructors
    public HashMap();
    public HashMap(int initialCapacity);
    public HashMap(Map m);
    public HashMap(int initialCapacity, float loadFactor);
// Methods Implementing Map
    public void clear();
    public boolean containsKey(Object key);
    public boolean containsValue(Object value);
    public Set entrySet();
    public Object get(Object key);
    public boolean isEmpty(); default:true
    public Set keySet();
    public Object put(Object key, Object value);
    public void putAll(Map t);
    public Object remove(Object key);
    public int size();
    public Collection values();
// Public Methods Overriding AbstractMap
    public Object clone();
}
  
```

*Subclasses:* LinkedHashMap, javax.print.attribute.standard.PrinterStateReasons

*Type Of:* java.beans.beancontext.BeanContextServicesSupport.services,  
java.beans.beancontext.BeanContextSupport.children

## HashSet

Java 1.2

java.util

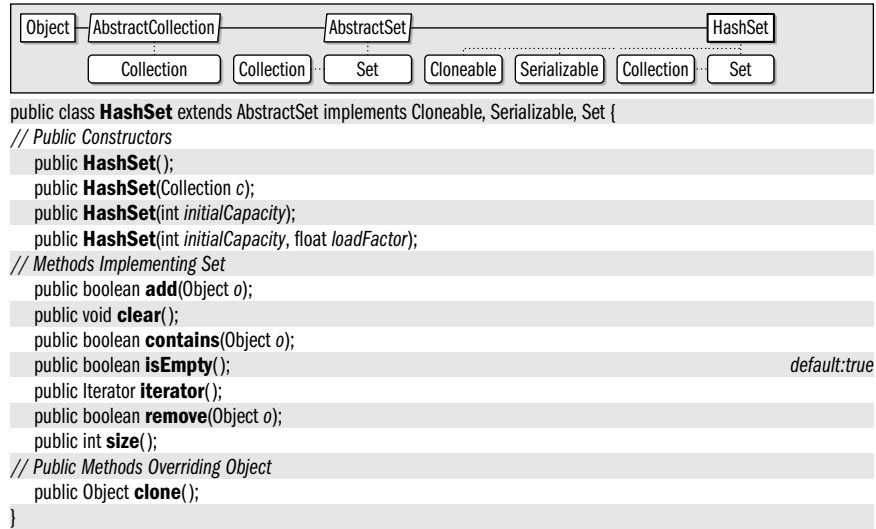
cloneable serializable collection

This class implements Set using an internal hashtable. It supports all optional Set and Collection methods and allows any type of object or null to be a member of the set. Because HashSet is based on a hashtable, the basic `add()`, `remove()`, and `contains()` methods are all quite efficient. HashSet makes no guarantee about the order in which the set elements are enumerated by the `Iterator` returned by `iterator()`. The methods of HashSet are not `synchronized`. If you are using it in a multithreaded environment, you must explicitly synchronize all code that modifies the set or obtain a `synchronized` wrapper for it by calling `Collections.synchronizedSet()`.

If you know in advance approximately how many mappings a HashSet will contain, you can improve efficiency by specifying *initialCapacity* when you call the `HashSet()` constructor. The *initialCapacity* argument times the *loadFactor* argument should be greater than the number of mappings the HashSet will contain. A good value for *loadFactor* is 0.75; this is also

## HashSet

the default value. See `Set` and `Collection` for details on the methods of `HashSet`. See also `TreeSet` and `HashMap`.



*Subclasses:* `LinkedHashSet`, `javax.print.attribute.standard.JobStateReasons`

## Hashtable

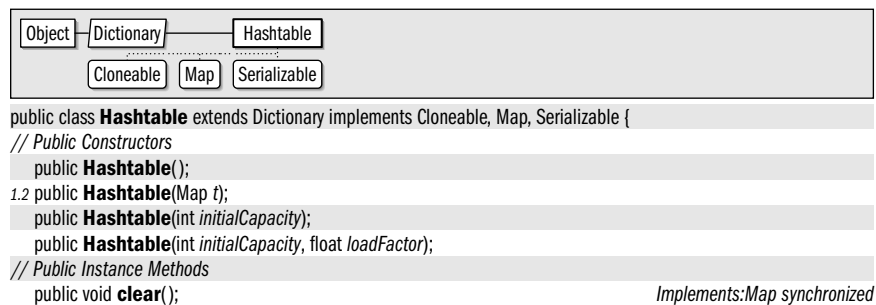
Java 1.0

`java.util`

*cloneable serializable collection*

This class implements a hashtable data structure, which maps key objects to value objects and allows the efficient lookup of the value associated with a given key. `put()` associates a value with a key in a `Hashtable`. `get()` retrieves a value for a specified key. `remove()` deletes a key/value association. `keys()` and `elements()` return `Enumeration` objects that allow you to iterate through the complete set of keys and values stored in the table. Objects used as keys in a `Hashtable` must have valid `equals()` and `hashCode()` methods (the versions inherited from `Object` are okay). `null` is not legal as a key or value in a `Hashtable`.

`Hashtable` is a commonly used class and has been a part of the Java API since Java 1.0. In Java 1.2, it has been enhanced to implement the `Map` interface, which defines some functionality in addition to the Java 1.0 `Hashtable` methods. `Hashtable` is very similar to the `HashMap` class, but has `synchronized` methods, which make it thread-safe but increase the overhead associated with it. If you need thread safety or require compatibility with Java 1.0 or Java 1.1, use `Hashtable`. Otherwise, use `HashMap`.



## IdentityHashMap

```
public boolean contains(Object value);                                synchronized
public boolean containsKey(Object key);                            Implements:Map synchronized
public Object get(Object key);                                    Implements:Map synchronized
public boolean isEmpty();                                         Implements:Map synchronized default:true
public Object put(Object key, Object value);                      Implements:Map synchronized
public Object remove(Object key);                                Implements:Map synchronized
public int size();                                                Implements:Map synchronized
// Methods Implementing Map
public void clear();                                              synchronized
public boolean containsKey(Object key);                            synchronized
1.2 public boolean containsValue(Object value);
1.2 public Set entrySet();
1.2 public boolean equals(Object o);                                synchronized
public Object get(Object key);                                    synchronized
1.2 public int hashCode();                                        synchronized
public boolean isEmpty();                                         synchronized default:true
1.2 public Set keySet();
public Object put(Object key, Object value);                      synchronized
1.2 public void putAll(Map t);                                    synchronized
public Object remove(Object key);                                synchronized
public int size();                                                synchronized
1.2 public Collection values();
// Public Methods Overriding Dictionary
public Enumeration elements();                                    synchronized
public Enumeration keys();                                        synchronized
// Public Methods Overriding Object
public Object clone();                                            synchronized
public String toString();                                        synchronized
// Protected Instance Methods
protected void rehash();
}
```

**Subclasses:** Properties, javax.swing.UIDefaults

**Passed To:** Too many methods to list.

**Returned By:** javax.naming.CannotProceedException.getEnvironment(),  
javax.naming.Context.getEnvironment(), javax.naming.InitialContext.getEnvironment(),  
javax.swing.JLayeredPane.getComponentToLayer(), javax.swing.JSlider.createStandardLabels()

**Type Of:** java.awt.GridBagLayout.comptable, java.text.RuleBasedBreakIterator.Builder.expressions,  
javax.naming.CannotProceedException.environment, javax.naming.InitialContext.myProps,  
javax.swing.JTable.{defaultEditorsByColumnClass, defaultRenderersByColumnClass},  
javax.swing.plaf.basic.BasicFileChooserUI.BasicFileView.iconCache,  
javax.swing.plaf.basic.BasicTreeUI.drawingCache, javax.swing.text.html.parser.DTD.{elementHash,  
entityHash}, javax.swing.undo.StateEdit.{postState, preState}

## IdentityHashMap

Java 1.4

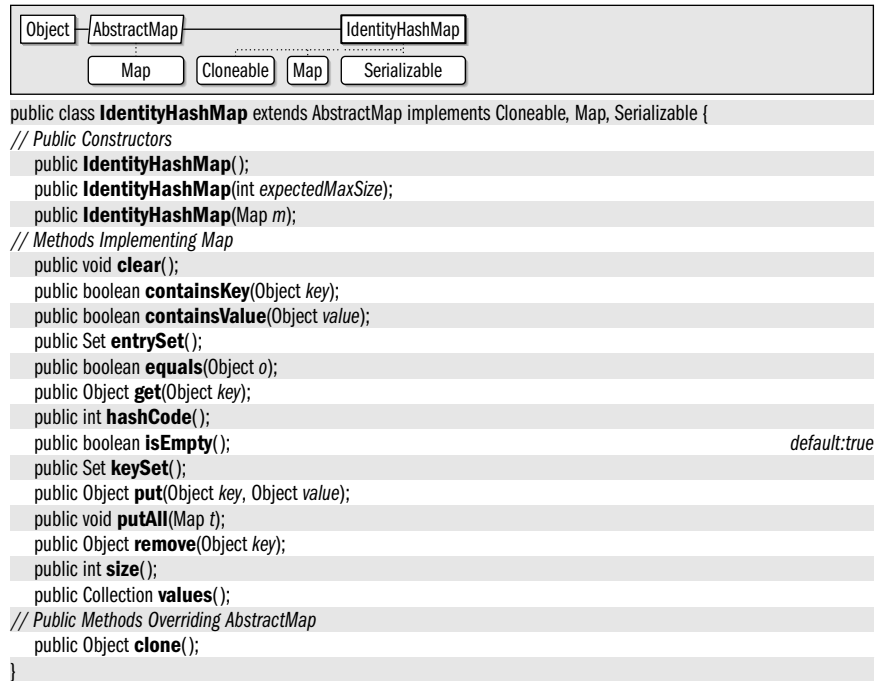
java.util

cloneable serializable collection

This Map implementation has an API similar to `HashMap` and uses an internal hashtable, as `HashMap` does. However, it behaves differently from `HashMap` in one very important way. When testing two keys to see if they are equal, `HashMap`, `LinkedHashMap`, and `TreeMap` use the `equals()` method to determine whether the two objects are indistinguishable in terms of their content or state. `IdentityHashMap` is different: it uses the `==` operator to determine whether the two key objects are identical—i.e., whether they are exactly the same object. This one difference in how key equality is tested has profound

## IdentityHashMap

ramifications for the behavior of the `Map`. In most cases, the equality testing of a `HashMap`, `LinkedHashMap`, or `TreeMap` is the appropriate behavior, and you should use one of those classes. For certain purposes, however, the identity testing of `IdentityHashMap` is required.



## Iterator

Java 1.2

### java.util

This interface defines methods for iterating, or enumerating, the elements of a collection. The `hasNext()` method returns `true` if there are more elements to be enumerated or `false` if all elements have already been returned. The `next()` method returns the next element. These two methods make it easy to loop through an iterator with code such as the following:

```
for(Iterator i = c.iterator(); i.hasNext(); )  
    processObject(i.next());
```

The `Iterator` interface is much like the `Enumeration` interface. In Java 1.2, `Iterator` is preferred over `Enumeration` because it provides a well-defined way to safely remove elements from a collection while the iteration is in progress. The `remove()` method removes the object most recently returned by `next()` from the collection that is being iterated through. Note, however, that support for `remove()` is optional; if an `Iterator` does not support `remove()`, it throws a `java.lang.UnsupportedOperationException` when you call it. While you are iterating through a collection, you are allowed to modify the collection only by calling the `remove()` method of the `Iterator`. If the collection is modified in any other way

while an iteration is ongoing, the `Iterator` may fail to operate correctly, or it may throw a `ConcurrentModificationException`.

```
public interface Iterator {
    // Public Instance Methods
    public abstract boolean hasNext();
    public abstract Object next();
    public abstract void remove();
}
```

*Implementations:* `java.beans.beancontext.BeanContextSupport.BCIterator`, `ListIterator`

*Passed To:* `javax.imageio.ImageReader.{getDestination(), readAll()}`,  
`javax.imageio.spi.ServiceRegistry.{registerServiceProviders(), ServiceRegistry()}`

*Returned By:* Too many methods to list.

## **LinkedHashMap**

**Java 1.4**

**java.util**

*cloneable serializable collection*

This class is a `Map` implementation based on a hashtable, just like its superclass `HashMap`. It defines no new public methods, and can be used exactly as `HashMap` is used. What is unique about this `Map` is that in addition to the hashtable data structure, it also uses a doubly-linked list to connect the keys of the `Map` into an internal list which defines a predictable iteration order.

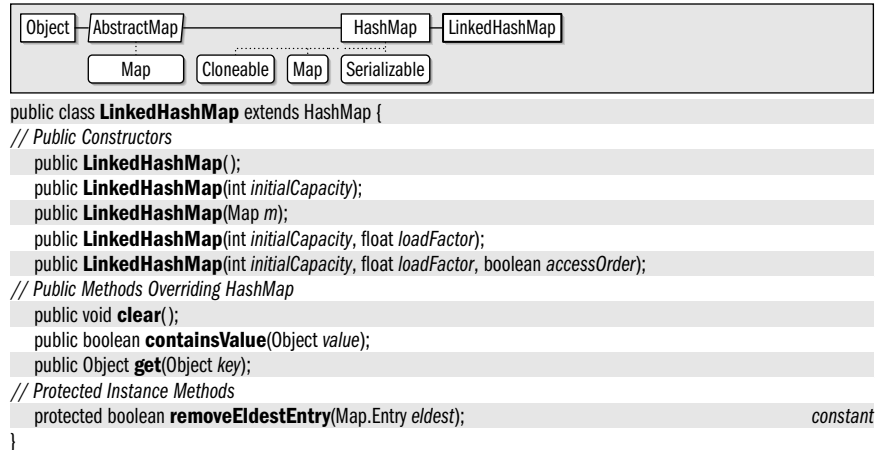
You can iterate through the keys or values of a `LinkedHashMap` by calling `entrySet()`, `keySet()`, or `values()` and then obtaining an `Iterator` for the returned collection, just as you would for a `HashMap`. When you do this, however, the keys and/or values are returned in a well-defined order rather than the essentially random order provided by a `HashMap`. The default ordering for `LinkedHashMap` is the insertion order of the key: the first key inserted into the `Map` is enumerated first (as is the value associated with it), and the last entry inserted is enumerated last. Note that this order is not affected by re-insertions. That is, if a `LinkedHashMap` contains a mapping from a key *k* to a value *v1*, and you call the `put()` method to map from *k* to a new value *v2*, this does not change the insertion order, or the iteration order of the key *k*. The iteration order of a value in the map is the iteration order of the key with which it is associated.

Insertion order is the default iteration order for this class, but if you instantiate a `LinkedHashMap` with the three-argument constructor, and pass `true` for the third argument, then the iteration order will be based on access order: the first key returned by an iterator is the one that was least-recently used in a `get()` or `put()` operation. The last key returned is the one that has been most-recently used. As with insertion order, the `values()` collection is iterated in the order defined by the keys with which those values are associated.

“Access ordering” is particularly useful for implementing LRU caches from which the Least-Recently Used elements are periodically purged. To facilitate this use, `LinkedHashMap` defines the protected `removeEldestEntry()` method. Each time the `put()` method is called (or for each mapping added by `putAll()`) the `LinkedHashMap` calls `removeEldestEntry()` and passes the least-recently used (or first inserted if insertion order is being used) `Map.Entry` object. If the method returns `true`, then that entry will be removed from the map. In `LinkedHashMap`, `removeEldestEntry()` always returns `false`, and old entries are never automatically removed, but you can override this behavior in a subclass. The decision to remove an old entry might be based on the content of the entry itself, or might more

## LinkedHashMap

simply be based on the `size()` of the `LinkedHashMap`. Note that `removeEldestEntry()` need simply return `true` or `false`; it should not remove the entry itself.



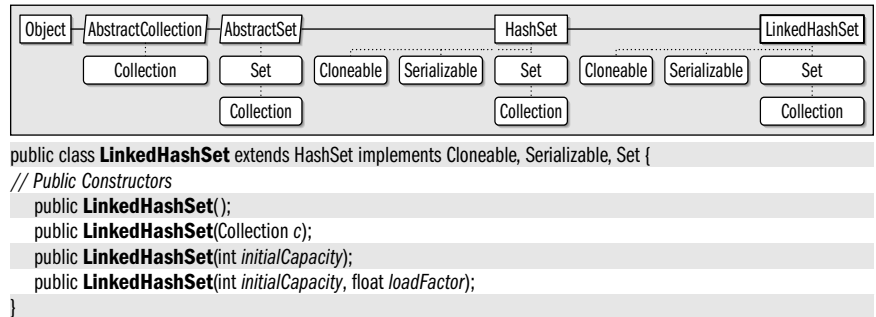
## LinkedHashSet

Java 1.4

java.util

cloneable serializable collection

This subclass of `HashSet` is a `Set` implementation based on a hashtable. It defines no new methods and is used just like a `HashSet`. What makes `LinkedHashSet` unique is that in addition to the hashtable data structure, it uses a doubly linked list to connect the elements of the set into an internal list in the order in which they were inserted. This means that the `Iterator` returned by the inherited `iterator()` method always enumerates the elements of the set in the order in which they were inserted. By contrast, the elements of a `HashSet` are enumerated in an essentially random order. Note that the iteration order is not affected by reinsertion of set elements. That is, if you attempt to add an element that already exists in the set, the iteration order of the set is not modified. If you delete an element and then reinsert it, the insertion order, and therefore the iteration order, does change.



## LinkedList

Java 1.2

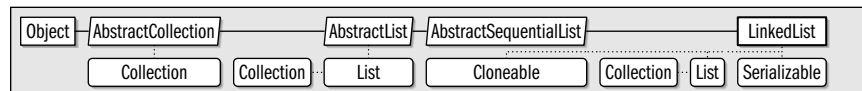
java.util

cloneable serializable collection

This class implements the `List` interface in terms of a doubly linked list. It supports all optional methods of `List` and `Collection` and allows list elements of any type, including null. Because `LinkedList` is implemented with a linked list data structure, the `get()` and `set()` methods are substantially less efficient than the same methods for an `ArrayList`. However,

a `LinkedList` may be more efficient when the `add()` and `remove()` methods are used frequently. The methods of `LinkedList` are not `synchronized`. If you are using a `LinkedList` in a multithreaded environment, you must explicitly synchronize any code that modifies the list or obtain a synchronized wrapper object with `Collections.synchronizedList()`.

In addition to the methods defined by the `List` interface, `LinkedList` defines methods to get the first and last elements of the list, to add an element to the beginning or end of the list, and to remove the first or last element of the list. These convenient and efficient methods make `LinkedList` well-suited for use as a stack or queue. See `List` and `Collection` for details on the methods of `LinkedList`. See also `ArrayList`.



```

public class LinkedList extends AbstractSequentialList implements Cloneable, java.util.List, Serializable {
    // Public Constructors
    public LinkedList();
    public LinkedList(Collection c);
    // Public Instance Methods
    public void addFirst(Object o);
    public void addLast(Object o);
    public Object getFirst();
    public Object getLast();
    public Object removeFirst();
    public Object removeLast();
    // Methods Implementing List
    public boolean add(Object o);
    public void add(int index, Object element);
    public boolean addAll(Collection c);
    public boolean addAll(int index, Collection c);
    public void clear();
    public boolean contains(Object o);
    public Object get(int index);
    public int indexOf(Object o);
    public int lastIndexOf(Object o);
    public ListIterator listIterator(int index);
    public boolean remove(Object o);
    public Object remove(int index);
    public Object set(int index, Object element);
    public int size();
    public Object[] toArray();
    public Object[] toArray(Object[] a);
    // Public Methods Overriding Object
    public Object clone();
}

```

## List

Java 1.2

java.util

collection

This interface represents an ordered collection of objects. Each element in a `List` has an index, or position, in the list, and elements can be inserted, queried, and removed by index. The first element of a `List` has an index of 0. The last element in a list has index `size()-1`.

In addition to the methods defined by the superinterface, `Collection`, `List` defines a number of methods for working with its indexed elements. `get()` and `set()` query and set the object at a particular index, respectively. Versions of `add()` and `addAll()` that take an `index`

## List

argument insert an object or `Collection` of objects at a specified index. The versions of `add()` and `addAll()` that do not take an *index* argument insert an object or collection of objects at the end of the list. `List` defines a version of `remove()` that removes the object at a specified index.

The `iterator()` method is just like the `iterator()` method of `Collection`, except that the `Iterator` it returns is guaranteed to enumerate the elements of the `List` in order. `listIterator()` returns a `ListIterator` object, which is more powerful than a regular `Iterator` and allows the list to be modified while iteration proceeds. `listIterator()` can take an *index* argument to specify where in the list iteration should begin.

`indexOf()` and `lastIndexOf()` perform linear searches from the beginning and end, respectively, of the list, searching for a specified object. Each method returns the index of the first matching object it finds, or `-1` if it does not find a match. Finally, `subList()` returns a `List` that contains only a specified contiguous range of list elements. The returned list is simply a view into the original list, so changes in the original `List` are visible in the returned `List`. This `subList()` method is particularly useful if you want to sort, search, `clear()`, or otherwise manipulate only a partial range of a larger list.

An interface cannot specify constructors, but it is conventional that all implementations of `List` provide at least two standard constructors: one that takes no arguments and creates an empty list, and a copy constructor that accepts an arbitrary `Collection` object that specifies the initial contents of the new `List`.

As with `Collection`, all `List` methods that change the contents of the list are optional, and implementations that do not support them simply throw `java.lang.UnsupportedOperationException`. Different implementations of `List` may have significantly different efficiency characteristics. For example, the `get()` and `set()` methods of an `ArrayList` are much more efficient than those of a `LinkedList`. On the other hand, the `add()` and `remove()` methods of a `LinkedList` can be more efficient than those of an `ArrayList`. See also `Collection`, `Set`, `Map`, `ArrayList`, and `LinkedList`.

Collection · List

```
public interface List extends Collection {
    // Public Instance Methods
    public abstract boolean add(Object o);
    public abstract void add(int index, Object element);
    public abstract boolean addAll(Collection c);
    public abstract boolean addAll(int index, Collection c);
    public abstract void clear();
    public abstract boolean contains(Object o);
    public abstract boolean containsAll(Collection c);
    public abstract boolean equals(Object o);
    public abstract Object get(int index);
    public abstract int hashCode();
    public abstract int indexOf(Object o);
    public abstract boolean isEmpty();
    public abstract Iterator iterator();
    public abstract int lastIndexOf(Object o);
    public abstract ListIterator listIterator();
    public abstract ListIterator listIterator(int index);
    public abstract boolean remove(Object o);
    public abstract Object remove(int index);
    public abstract boolean removeAll(Collection c);
    public abstract boolean retainAll(Collection c);
    public abstract Object set(int index, Object element);
```



```

    public abstract int size();
    public abstract java.util.List subList(int fromIndex, int toIndex);
    public abstract Object[] toArray();
    public abstract Object[] toArray(Object[] a);
}

```

*Implementations:* AbstractList, ArrayList, LinkedList, Vector

*Passed To:* Too many methods to list.

*Returned By:* Too many methods to list.

*Type Of:* Collections.EMPTY\_LIST, javax.imageio.IIOImage.thumbnails,  
 javax.imageio.ImageReader.{progressListeners, updateListeners, warningListeners, warningLocales},  
 javax.imageio.ImageWriter.{progressListeners, warningListeners, warningLocales}

## ListIterator

Java 1.2

java.util

This interface is an extension of `Iterator` for use with ordered collections, or lists. It defines methods to iterate forward and backward through a list, to determine the list index of the elements being iterated, and, for mutable lists, to safely insert, delete, and edit elements in the list while the iteration is in progress. For some lists, notably `LinkedList`, using an iterator to enumerate the list's elements may be substantially more efficient than looping through the list by index and calling `get()` repeatedly.

`hasNext()` and `next()` are the most commonly used methods of `ListIterator`; they iterate forward through the list. See `Iterator` for details. In addition to these two methods, however, `ListIterator` also defines `hasPrevious()` and `previous()` that allow you to iterate backward through the list. `previous()` returns the previous element on the list or throws a `NoSuchElementException` if there is no previous element. `hasPrevious()` returns `true` if a subsequent call to `previous()` returns an object. `nextIndex()` and `previousIndex()` return the index of the object that would be returned by a subsequent call to `next()` or `previous()`. If `next()` or `previous()` throw a `NoSuchElementException`, `nextIndex()` returns the size of the list, and `previousIndex()` returns `-1`.

`ListIterator` defines three optionally supported methods that provide a safe way to modify the contents of the underlying list while the iteration is in progress. `add()` inserts a new object into the list, immediately before the object that would be returned by a subsequent call to `next()`. Calling `add()` does not affect the value that is returned by `next()`, however. If you call `previous()` immediately after calling `add()`, the method returns the object you just added. `remove()` deletes from the list the object most recently returned by `next()` or `previous()`. You can only call `remove()` once per call to `next()` or `previous()`. If you have called `add()`, you must call `next()` or `previous()` again before calling `remove()`. `set()` replaces the object most recently returned by `next()` or `previous()` with the specified object. If you have called `add()` or `remove()`, you must call `next()` or `previous()` again before calling `set()`. Remember that support for the `add()`, `remove()`, and `set()` methods is optional. Iterators for immutable lists never support them, of course. An unsupported method throws a `java.lang.UnsupportedOperationException` when called. Also, when an iterator is in use, all modifications should be made through the iterator rather than to the list itself. If the underlying list is modified while an iteration is ongoing, the `ListIterator` may fail to operate correctly or may throw a `ConcurrentModificationException`.

Iterator ··· ListIterator

```

public interface ListIterator extends Iterator {
    // Public Instance Methods

```

## ListIterator

```
public abstract void add(Object o);
public abstract boolean hasNext();
public abstract boolean hasPrevious();
public abstract Object next();
public abstract int nextIndex();
public abstract Object previous();
public abstract int previousIndex();
public abstract void remove();
public abstract void set(Object o);
}
```

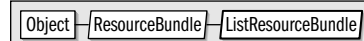
*Returned By:* AbstractList.listIterator(), AbstractSequentialList.listIterator(), LinkedList.listIterator(), java.util.List.listIterator()

## ListResourceBundle

Java 1.1

java.util

This abstract class provides a simple way to define a `ResourceBundle`. You may find it easier to subclass `ListResourceBundle` than to subclass `ResourceBundle` directly. `ListResourceBundle` provides implementations for the abstract `handleGetObject()` and `getKeys()` methods defined by `ResourceBundle` and adds its own abstract `getContents()` method a subclass must override. `getContents()` returns an `Object[][]`—an array of arrays of objects. This array can have any number of elements. Each element of this array must itself be an array with two elements: the first element of each subarray should be a `String` that specifies the name of a resource, and the corresponding second element should be the value of that resource; this value can be an `Object` of any desired type. See also `ResourceBundle` and `PropertyResourceBundle`.



```
public abstract class ListResourceBundle extends ResourceBundle {
// Public Constructors
    public ListResourceBundle();
// Public Methods Overriding ResourceBundle
    public Enumeration getKeys();
    public final Object handleGetObject(String key);
// Protected Instance Methods
    protected abstract Object[][] getContents();
}
```

*Subclasses:* javax.accessibility.AccessibleResourceBundle

## Locale

Java 1.1

java.util

*cloneable serializable*

The `Locale` class represents a locale: a political, geographical, or cultural region that typically has a distinct language and distinct customs and conventions for such things as formatting dates, times, and numbers. The `Locale` class defines a number of constants that represent commonly used locales. `Locale` also defines a static `getDefault()` method that returns the default `Locale` object, which represents a locale value inherited from the host system. `getAvailableLocales()` returns the list of all locales supported by the underlying system. If none of these methods for obtaining a `Locale` object are suitable, you can explicitly create your own `Locale` object. To do this, you must specify a language code and optionally a country code and variant string. `getISOCountries()` and `getISOLanguages()` return the list of supported country codes and language codes.

The `Locale` class does not implement any internationalization behavior itself; it merely serves as a locale identifier for those classes that can localize their behavior. Given a `Locale` object, you can invoke the various `getDisplay` methods to obtain a description of the locale suitable for display to a user. These methods may themselves take a `Locale` argument, so the names of languages and countries can be localized as appropriate.



## Locale

```
// Public Methods Overriding Object
public Object clone();
public boolean equals(Object obj);
public int hashCode(); synchronized
public final String toString();
}
```

*Passed To:* Too many methods to list.

*Returned By:* Too many methods to list.

*Type Of:* Too many fields to list.

## Map

Java 1.2

java.util

collection

This interface represents a collection of mappings, or associations, between key objects and value objects. Hashtables and associative arrays are examples of maps. The set of key objects in a **Map** must not have any duplicates; the collection of value objects is under no such constraint. The key objects should usually be immutable objects, or, if they are not, care should be taken that they do not change while in use in a **Map**. As of Java 1.2, the **Map** interface replaces the abstract **Dictionary** class. Although a **Map** is not a **Collection**, the **Map** interface is still considered an integral part, along with **Set**, **List**, and others, of the Java collections framework.

You can add a key/value association to a **Map** with the **put()** method. Use **putAll()** to copy all mappings from one **Map** to another. Call **get()** to look up the value object associated with a specified key object. Use **remove()** to delete the mapping between a specified key and its value, or use **clear()** to delete all mappings from a **Map**. **size()** returns the number of mappings in a **Map**, and **isEmpty()** tests whether the **Map** contains no mappings. **containsKey()** tests whether a **Map** contains the specified key object, and **containsValue()** tests whether it contains the specified value. (For most implementations, **containsValue()** is a much more expensive operation than **containsKey()**, however.) **keySet()** returns a **Set** of all key objects in the **Map**. **values()** returns a **Collection** (not a **Set**, since it may contain duplicates) of all value objects in the map. **entrySet()** returns a **Set** of all mappings in a **Map**. The elements of this returned **Set** are **Map.Entry** objects. The collections returned by **values()**, **keySet()**, and **entrySet()** are based on the **Map** itself, so changes to the **Map** are reflected in the collections.

An interface cannot specify constructors, but it is conventional that all implementations of **Map** provide at least two standard constructors: one that takes no arguments and creates an empty map, and a copy constructor that accepts a **Map** object that specifies the initial contents of the new **Map**.

Implementations are required to support all methods that query the contents of a **Map**, but support for methods that modify the contents of a **Map** is optional. If an implementation does not support a particular method, the implementation of that method simply throws a **java.lang.UnsupportedOperationException**. See also **Collection**, **Set**, **List**, **HashMap**, **Hashtable**, **WeakHashMap**, **SortedMap**, and **TreeMap**.

```
public interface Map {
// Inner Classes
    public static interface Entry;
// Public Instance Methods
    public abstract void clear();
    public abstract boolean containsKey(Object key);
    public abstract boolean containsValue(Object value);
}
```

```

public abstract Set entrySet();
public abstract boolean equals(Object o);
public abstract Object get(Object key);
public abstract int hashCode();
public abstract boolean isEmpty();
public abstract Set keySet();
public abstract Object put(Object key, Object value);
public abstract void putAll(Map t);
public abstract Object remove(Object key);
public abstract int size();
public abstract Collection values();
}

```

**Implementations:** java.awt.RenderingHints, AbstractMap, HashMap, Hashtable, IdentityHashMap, SortedMap, WeakHashMap, java.util.jar.Attributes

**Passed To:** Too many methods to list.

**Returned By:** Too many methods to list.

**Type Of:** java.awt.Toolkit.desktopProperties, Collections.EMPTY\_MAP, java.util.jar.Attributes.map

## Map.Entry

Java 1.2

java.util

This interface represents a single mapping, or association, between a key object and a value object in a Map. The `entrySet()` method of a Map returns a Set of Map.Entry objects that represent the set of mappings in the map. Use the `iterator()` method of that Set to enumerate these Map.Entry objects. Use `getKey()` and `getValue()` to obtain the key and value objects for the entry. Use the optionally supported `setValue()` method to change the value of an entry. This method throws a `java.lang.UnsupportedOperationException` if it is not supported by the implementation.

```

public static interface Map.Entry {
// Public Instance Methods
public abstract boolean equals(Object o);
public abstract Object getKey();
public abstract Object getValue();
public abstract int hashCode();
public abstract Object setValue(Object value);
}

```

**Passed To:** LinkedHashMap.removeEldestEntry()

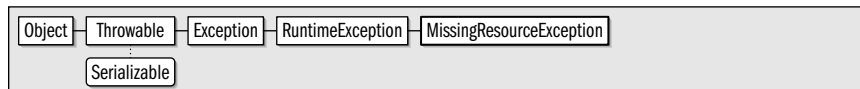
## MissingResourceException

Java 1.1

java.util

serializable unchecked

This signals that no ResourceBundle can be located for the desired locale or that a named resource cannot be found within a given ResourceBundle. `getClassName()` returns the name of the ResourceBundle class in question, and `getKey()` returns the name of the resource that cannot be located.



```

public class MissingResourceException extends RuntimeException {
// Public Constructors

```

## *MissingResourceException*

```
public MissingResourceException(String s, String className, String key);  
// Public Instance Methods  
public String getClassName();  
public String getKey();  
}
```

*Thrown By:* Locale.{getISO3Country(), getISO3Language()}

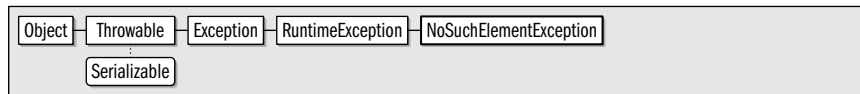
## **NoSuchElementException**

**Java 1.0**

java.util

*serializable unchecked*

This signals that there are no elements in an object (such as a `Vector`) or that there are no more elements in an object (such as an `Enumeration`).



```
public class NoSuchElementException extends RuntimeException {  
// Public Constructors  
public NoSuchElementException();  
public NoSuchElementException(String s);  
}
```

## **Observable**

**Java 1.0**

java.util

This class is the superclass for classes that want to provide notifications of state changes to interested `Observer` objects. Register an `Observer` to be notified by passing it to the `addObserver()` method of an `Observable` and deregister it by passing it to the `deleteObserver()` method. You can delete all observers registered for an `Observable` with `deleteObservers()` and find out how many observers have been added with `countObservers()`. Note that there is not a method to enumerate the particular `Observer` objects that have been added.

An `Observable` subclass should call the protected method `setChanged()` when its state has changed in some way. This sets a "state changed" flag. After an operation or series of operations that may have caused the state to change, the `Observable` subclass should call `notifyObservers()`, optionally passing an arbitrary `Object` argument. If the state changed flag is set, this `notifyObservers()` calls the `update()` method of each registered `Observer` (in some arbitrary order), passing the `Observable` object and the optional argument, if any. Once the `update()` method of each `Observable` has been called, `notifyObservers()` calls `clearChanged()` to clear the state changed flag. If `notifyObservers()` is called when the state changed flag is not set, it does not do anything. You can use `hasChanged()` to query the current state of the changed flag.

The `Observable` class and `Observer` interface are not commonly used. Most applications prefer the event-based notification model defined by the JavaBeans component framework and by the `EventObject` class and `EventListener` interface of this package.

```
public class Observable {  
// Public Constructors  
public Observable();  
// Public Instance Methods  
public void addObserver(Observer o); synchronized  
public int countObservers(); synchronized  
public void deleteObserver(Observer o); synchronized
```

```

public void deleteObservers(); synchronized
public boolean hasChanged(); synchronized
public void notifyObservers();
public void notifyObservers(Object arg);
// Protected Instance Methods
protected void clearChanged(); synchronized
protected void setChanged(); synchronized
}

```

*Passed To:* Observer.update()

## Observer

Java 1.0

java.util

This interface defines the `update()` method required for an object to observe subclasses of `Observable`. An `Observer` registers interest in an `Observable` object by calling the `addObserver()` method of `Observable`. `Observer` objects that have been registered in this way have their `update()` methods invoked by the `Observable` when that object has changed.

This interface is conceptually similar to, but less commonly used than, the `EventListener` interface and its various event-specific subinterfaces.

```

public interface Observer {
// Public Instance Methods
    public abstract void update(Observable o, Object arg);
}

```

*Passed To:* Observable.{addObserver(), deleteObserver()}

## Properties

Java 1.0

java.util

*cloneable serializable collection*

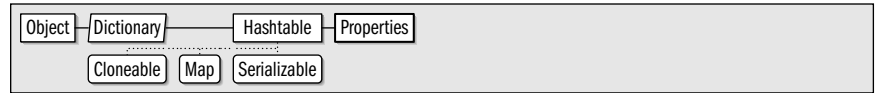
This class is an extension of `Hashtable` that allows key/value pairs to be read from and written to a stream. The `Properties` class implements the system properties list, which supports user customization by allowing programs to look up the values of named resources. Because the `load()` and `store()` methods provide an easy way to read and write properties from and to a text stream, this class provides a convenient way to implement an application configuration file.

When you create a `Properties` object, you may specify another `Properties` object that contains default values. Keys (property names) and values are associated in a `Properties` object with the `Hashtable` method `put()`. Values are looked up with `getProperty()`; if this method does not find the key in the current `Properties` object, it looks in the default `Properties` object that was passed to the constructor method. A default value can also be specified, in case the key is not found at all. Use `setProperty()` to add a property name/value pair to the `Properties` object. This Java 1.2 method is preferred over the inherited `put()` method because it enforces the constraint that property names and values be strings.

`propertyNames()` returns an enumeration of all property names (keys) stored in the `Properties` object and (recursively) all property names stored in the default `Properties` object associated with it. `list()` prints the properties stored in a `Properties` object, which can be useful for debugging. `store()` writes a `Properties` object to a stream, writing one property per line, in name=value format. As of Java 1.2, `store()` is preferred over the deprecated `save()` method, which writes properties in the same way but suppresses any I/O exceptions that may be thrown in the process. The second argument to both `store()` and `save()` is a comment that is written out at the beginning of the property file. Finally, `load()`

## Properties

reads key/value pairs from a stream and stores them in a `Properties` object. It is suitable for reading both properties written with `store()` and hand-edited properties files.



```
public class Properties extends Hashtable {
    // Public Constructors
    public Properties();
    public Properties(Properties defaults);
    // Public Instance Methods
    public String getProperty(String key);
    public String getProperty(String key, String defaultValue);
    1.1 public void list(java.io.PrintWriter out);
    public void list(java.io.PrintStream out);
    public void load(java.io.InputStream inStream) throws java.io.IOException;           synchronized
    public Enumeration propertyNames();
    1.2 public Object setProperty(String key, String value);           synchronized
    1.2 public void store(java.io.OutputStream out, String header) throws java.io.IOException; synchronized
    // Protected Instance Fields
    protected Properties defaults;
    // Deprecated Public Methods
    # public void save(java.io.OutputStream out, String header);           synchronized
}
```

*Subclasses:* `java.security.Provider`

*Passed To:* `java.awt.Toolkit.getPrintJob()`, `System.setProperties()`,  
`java.rmi.activation.ActivationGroupDesc.ActivationGroupDesc()`, `java.sql.Driver.{connect(),  
getPropertyInfo()}, java.sql.DriverManager.getConnection(), Properties.Properties(),  
javax.naming.CompoundName.CompoundName(),  
javax.xml.transform.Transformer.setOutputProperties(), org.omg.CORBA.ORB.{init(), set_parameters()}`

*Returned By:* `System.getProperties()`,  
`java.rmi.activation.ActivationGroupDesc.getPropertyOverrides()`,  
`javax.xml.transform.Templates.getOutputProperties()`,  
`javax.xml.transform.Transformer.getOutputProperties()`

*Type Of:* `Properties.defaults`, `javax.naming.CompoundName.mySyntax`

## PropertyPermission

**Java 1.2**

`java.util`

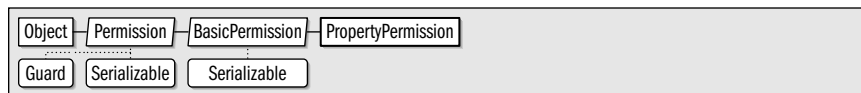
*serializable permission*

This class is a `java.security.Permission` that governs read and write access to system properties with `System.getProperty()` and `System.setProperty()`. A `PropertyPermission` object has a name, or target, and a comma-separated list of actions. The name of the permission is the name of the property of interest. The action string can be “read” for `getProperty()` access, “write” for `setProperty()` access, or “read,write” for both types of access. `PropertyPermission` extends `java.security.BasicPermission`, so the name of the property supports simple wildcards. The name “\*” represents any property name. If a name ends with “.\*”, it represents any property names that share the specified prefix. For example, the name “java.\*” represents “java.version”, “java.vendor”, “java.vendor.url”, and all other properties that begin with “java”.

Granting access to system properties is not overtly dangerous, but caution is still necessary. Some properties, such as “user.home”, reveal details about the host system that malicious code can use to mount an attack. Programmers writing system-level code and



system administrators configuring security policies may need to use this class, but applications never need to use it.



```

public final class PropertyPermission extends java.security.BasicPermission {
// Public Constructors
    public PropertyPermission(String name, String actions);
// Public Methods Overriding BasicPermission
    public boolean equals(Object obj);
    public String getActions();
    public int hashCode();
    public boolean implies(java.security.Permission p);
    public java.security.PermissionCollection newPermissionCollection();
}

```

## PropertyResourceBundle

Java 1.1

java.util

This class is a concrete subclass of `ResourceBundle`. It reads a `Properties` file from a specified `InputStream` and implements the `ResourceBundle` API for looking up named resources from the resulting `Properties` object. A `Properties` file contains lines of the form:

*name=value*

Each such line defines a named property with the specified `String` value. Although you can instantiate a `PropertyResourceBundle` yourself, it is more common to simply define a `Properties` file and then allow `ResourceBundle.getBundle()` to look up that file and return the necessary `PropertyResourceBundle` object. See also `Properties` and `ResourceBundle`.



```

public class PropertyResourceBundle extends ResourceBundle {
// Public Constructors
    public PropertyResourceBundle(java.io.InputStream stream) throws java.io.IOException;
// Public Methods Overriding ResourceBundle
    public Enumeration getKeys();
    public Object handleGetObject(String key);
}

```

## Random

Java 1.0

java.util

serializable

This class implements a pseudo-random number generator suitable for games and similar applications. If you need a cryptographic-strength source of pseudo-randomness, see `java.security.SecureRandom`. `nextDouble()` and `nextFloat()` return a value between 0.0 and 1.0. `nextLong()` and the no-argument version of `nextInt()` return long and int values distributed across the range of those data types. In Java 1.2, if you pass an argument to `nextInt()`, it returns a value between zero (inclusive) and the specified number (exclusive). `nextGaussian()` returns pseudo-random floating-point values with a Gaussian distribution; the mean of the values is 0.0 and the standard deviation is 1.0. `nextBoolean()` returns a pseudo-random boolean value, and `nextBytes()` fills in the specified byte array with pseudo-random bytes. You can use the `setSeed()` method or the optional constructor argument to initialize the pseudo-random number generator with some variable

## Random

seed value other than the current time (the default) or with a constant to ensure a repeatable sequence of pseudo-randomness.

```
Object Random Serializable

public class Random implements Serializable {
// Public Constructors
    public Random();
    public Random(long seed);
// Public Instance Methods
    1.2 public boolean nextBoolean();
    1.1 public void nextBytes(byte[] bytes);
    public double nextDouble();
    public float nextFloat();
    public double nextGaussian();
    public int nextInt();
    1.2 public int nextInt(int n);
    public long nextLong();
    public void setSeed(long seed);
// Protected Instance Methods
    1.1 protected int next(int bits);
}
```

*Subclasses:* java.security.SecureRandom

*Passed To:* java.math.BigInteger.{BigInteger(), probablePrime()}, Collections.shuffle()

## RandomAccess

Java 1.4

java.util

This marker interface is implemented by List implementations to advertise that they provide efficient random access (usually constant time) to all list elements. ArrayList and Vector implement this interface, but LinkedList does not. Classes that manipulate generic List objects may want to test for this interface with instanceof and use different algorithms for lists that provide efficient random access than they use for lists that are most efficiently accessed sequentially.

```
public interface RandomAccess {
}
```

*Implementations:* ArrayList, Vector

## ResourceBundle

Java 1.1

java.util

This abstract class allows subclasses to define sets of localized resources that can then be dynamically loaded as needed by internationalized programs. Such resources may include user-visible text and images that appear in an application, as well as more complex things such as Menu objects. Use getBundle() to load a ResourceBundle subclass that is appropriate for the default or specified locale. Use getObject(), getString(), and getStringArray() to look up a named resource in a bundle. To define a bundle, provide implementations of handleGetObject() and getKeys(). It is often easier, however, to subclass ListResourceBundle or provide a Properties file that is used by PropertyResourceBundle. The name of any localized ResourceBundle class you define should include the locale language code, and, optionally, the locale country code.

```
public abstract class ResourceBundle {
// Public Constructors
```

```

    public ResourceBundle();
// Public Class Methods
    public static final ResourceBundle getBundle(String baseName);
    public static final ResourceBundle getBundle(String baseName, Locale locale);
1.2 public static ResourceBundle getBundle(String baseName, Locale locale, ClassLoader loader);
// Public Instance Methods
    public abstract Enumeration getKeys();
1.2 public Locale getLocale();
    public final Object getObject(String key);
    public final String getString(String key);
    public final String[] getStringArray(String key);
// Protected Instance Methods
    protected abstract Object handleGetObject(String key);
    protected void setParent(ResourceBundle parent);
// Protected Instance Fields
    protected ResourceBundle parent;
}

```

**Subclasses:** ListResourceBundle, PropertyResourceBundle

**Passed To:** java.awt.ComponentOrientation.getOrientation(), java.awt.Window.applyResourceBundle(), ResourceBundle.setParent(), java.util.logging.LogRecord.setResourceBundle()

**Returned By:** ResourceBundle.getBundle(), java.util.logging.Logger.getResourceBundle(), java.util.logging.LogRecord.getResourceBundle()

**Type Of:** ResourceBundle.parent

## Set

Java 1.2

java.util

collection

This interface represents an unordered **Collection** of objects that contains no duplicate elements. That is, a **Set** cannot contain two elements **e1** and **e2** where **e1.equals(e2)**, and it can contain at most one null element. The **Set** interface defines the same methods as its superinterface, **Collection**. It constrains the **add()** and **addAll()** methods from adding duplicate elements to the **Set**.

An interface cannot specify constructors, but it is conventional that all implementations of **Set** provide at least two standard constructors: one that takes no arguments and creates an empty set, and a copy constructor that accepts a **Collection** object that specifies the initial contents of the new **Set**. This copy constructor must ensure that duplicate elements are not added to the **Set**, of course.

As with **Collection**, the **Set** methods that modify the contents of the set are optional, and implementations that do not support these methods simply throw **java.lang.UnsupportedOperationException**. See also **Collection**, **List**, **Map**, **SortedSet**, **HashSet**, and **TreeSet**.

Collection · Set

```

public interface Set extends Collection {
// Public Instance Methods
    public abstract boolean add(Object o);
    public abstract boolean addAll(Collection c);
    public abstract void clear();
    public abstract boolean contains(Object o);
    public abstract boolean containsAll(Collection c);
    public abstract boolean equals(Object o);
    public abstract int hashCode();
}

```

## Set

```
public abstract boolean isEmpty();
public abstract Iterator iterator();
public abstract boolean remove(Object o);
public abstract boolean removeAll(Collection c);
public abstract boolean retainAll(Collection c);
public abstract int size();
public abstract Object[] toArray();
public abstract Object[] toArray(Object[] a);
}
```

**Implementations:** AbstractSet, HashSet, LinkedHashSet, SortedSet

**Passed To:** Too many methods to list.

**Returned By:** Too many methods to list.

**Type Of:** Collections.EMPTY\_SET

## SimpleTimeZone

Java 1.1

java.util

cloneable serializable

This concrete subclass of `TimeZone` is a simple implementation of that abstract class that is suitable for use in locales that use the Gregorian calendar. Programs do not normally need to instantiate this class directly; instead, they use one of the static factory methods of `TimeZone` to obtain a suitable `TimeZone` subclass. The only reason to instantiate this class directly is if you need to support a time zone with non-standard-daylight-savings-time rules. In that case, you can call `setStartRule()` and `setEndRule()` to specify the starting and ending dates of daylight-savings time for the time zone.



```
public class SimpleTimeZone extends TimeZone {
// Public Constructors
    public SimpleTimeZone(int rawOffset, String ID);
    public SimpleTimeZone(int rawOffset, String ID, int startMonth, int startDay, int startDayOfWeek, int startTime,
        int endMonth, int endDay, int endDayOfWeek, int endTime);
1.2 public SimpleTimeZone(int rawOffset, String ID, int startMonth, int startDay, int startDayOfWeek, int startTime,
    int endMonth, int endDay, int endDayOfWeek, int endTime, int dstSavings);
1.4 public SimpleTimeZone(int rawOffset, String ID, int startMonth, int startDay, int startDayOfWeek, int startTime,
    int startTimeMode, int endMonth, int endDay, int endDayOfWeek, int endTime,
    int endTimeMode, int dstSavings);

// Public Constants
1.4 public static final int STANDARD_TIME; =1
1.4 public static final int UTC_TIME; =2
1.4 public static final int WALL_TIME; =0
// Public Instance Methods
1.2 public void setDSTSavings(int millisSavedDuringDST);
1.2 public void setEndRule(int endMonth, int endDay, int endTime);
    public void setEndRule(int endMonth, int endDay, int endDayOfWeek, int endTime);
1.2 public void setEndRule(int endMonth, int endDay, int endDayOfWeek, int endTime, boolean after);
1.2 public void setStartRule(int startMonth, int startDay, int startTime);
    public void setStartRule(int startMonth, int startDay, int startDayOfWeek, int startTime);
1.2 public void setStartRule(int startMonth, int startDay, int startDayOfWeek, int startTime, boolean after);
    public void setStartYear(int year);
// Public Methods Overriding TimeZone
```

```

    public Object clone();
1.2 public int getDSTSavings();
1.4 public int getOffset(long date);
    public int getOffset(int era, int year, int month, int day, int dayOfWeek, int millis);
    public int getRawOffset();
1.2 public boolean hasSameRules(TimeZone other);
    public boolean inDaylightTime(java.util.Date date);
    public void setRawOffset(int offsetMillis);
    public boolean useDaylightTime();
// Public Methods Overriding Object
    public boolean equals(Object obj);
    public int hashCode(); synchronized
    public String toString();
}

```

**SortedMap****Java 1.2**

java.util

collection

This interface represents a **Map** object that keeps its set of key objects in sorted order. As with **Map**, it is conventional that all implementations of this interface define a no-argument constructor to create an empty map and a copy constructor that accepts a **Map** object that specifies the initial contents of the **SortedMap**. Furthermore, when creating a **SortedMap**, there should be a way to specify a **Comparator** object to sort the key objects of the map. If no **Comparator** is specified, all key objects must implement the **java.lang.Comparable** interface so they can be sorted in their natural order. See also **Map**, **TreeMap**, and **SortedSet**.

The inherited **keySet()**, **values()**, and **entrySet()** methods return collections that can be iterated in the sorted order. **firstKey()** and **lastKey()** return the lowest and highest key values in the **SortedMap**. **subMap()** returns a **SortedMap** that contains only mappings for keys from (and including) the first specified key up to (but not including) the second specified key. **headMap()** returns a **SortedMap** that contains mappings whose keys are less than (but not equal to) the specified key. **tailMap()** returns a **SortedMap** that contains mappings whose keys are greater than or equal to the specified key. **subMap()**, **headMap()**, and **tailMap()** return **SortedMap** objects that are simply views of the original **SortedMap**; any changes in the original map are reflected in the returned map and vice versa.

Map
 SortedMap

```

public interface SortedMap extends Map {
// Public Instance Methods
    public abstract Comparator comparator();
    public abstract Object firstKey();
    public abstract SortedMap headMap(Object toKey);
    public abstract Object lastKey();
    public abstract SortedMap subMap(Object fromKey, Object toKey);
    public abstract SortedMap tailMap(Object fromKey);
}

```

*Implementations:* **TreeMap***Passed To:* **Collections**.{**synchronizedSortedMap()**, **unmodifiableSortedMap()**}, **TreeMap**.**TreeMap()***Returned By:* **java.nio.charset.Charset.availableCharsets()**, **Collections**.{**synchronizedSortedMap()**, **unmodifiableSortedMap()**}, **SortedMap**.{**headMap()**, **subMap()**, **tailMap()**}, **TreeMap**.{**headMap()**, **subMap()**, **tailMap()**}

## SortedSet

### SortedSet

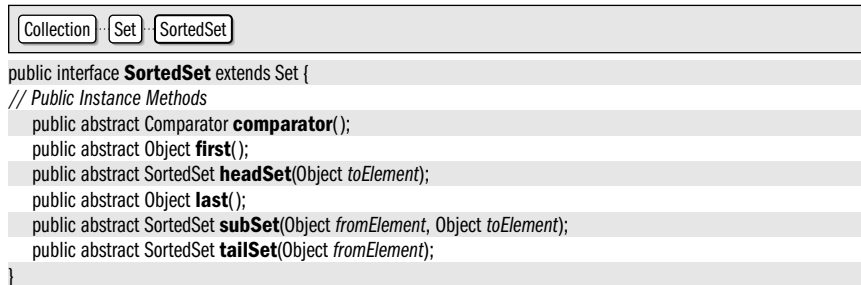
Java 1.2

java.util

collection

This interface is a `Set` that sorts its elements and guarantees that its `iterator()` method returns an `Iterator` that enumerates the elements of the set in sorted order. As with the `Set` interface, it is conventional for all implementations of `SortedSet` to provide a no-argument constructor that creates an empty set and a copy constructor that expects a `Collection` object specifying the initial (unsorted) contents of the set. Furthermore, when creating a `SortedSet`, there should be a way to specify a `Comparator` object that compares and sorts the elements of the set. If no `Comparator` is specified, the elements of the set must all implement `java.lang.Comparable` so they can be sorted in their natural order. See also `Set`, `TreeSet`, and `SortedMap`.

`SortedSet` defines a few methods in addition to those it inherits from the `Set` interface. `first()` and `last()` return the lowest and highest objects in the set. `headSet()` returns all elements from the beginning of the set up to (but not including) the specified element. `tailSet()` returns all elements between (and including) the specified element and the end of the set. `subSet()` returns all elements of the set from (and including) the first specified element up to (but excluding) the second specified element. Note that all three methods return a `SortedSet` that is implemented as a view onto the original `SortedSet`. Changes in the original set are visible through the returned set and vice versa.



**Implementations:** `TreeSet`

**Passed To:** `Collections.{synchronizedSortedSet(), unmodifiableSortedSet()}`, `TreeSet.TreeSet()`

**Returned By:** `Collections.{synchronizedSortedSet(), unmodifiableSortedSet()}`, `SortedSet.{headSet(), subSet(), tailSet()}`, `TreeSet.{headSet(), subSet(), tailSet()}`

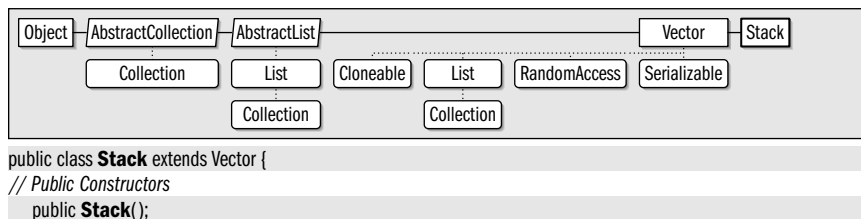
### Stack

Java 1.0

java.util

cloneable serializable collection

This class implements a last-in-first-out (LIFO) stack of objects. `push()` puts an object on the top of the stack. `pop()` removes and returns the top object from the stack. `peek()` returns the top object without removing it. In Java 1.2, you can instead use a `LinkedList` as a stack.



```
// Public Instance Methods
public boolean empty();
public Object peek();                                synchronized
public Object pop();                                  synchronized
public Object push(Object item);
public int search(Object o);                          synchronized
}
```

Type Of: java.text.RuleBasedBreakIterator.Builder.decisionPointStack

## StringTokenizer

Java 1.0

java.util

When a `StringTokenizer` is instantiated with a `String`, it breaks the string up into tokens separated by any of the characters in the specified string of delimiters. (For example, words separated by space and tab characters are tokens.) The `hasMoreTokens()` and `nextToken()` methods obtain the tokens in order. `countTokens()` returns the number of tokens in the string. `StringTokenizer` implements the `Enumeration` interface, so you may also access the tokens with the familiar `hasMoreElements()` and `nextElement()` methods. When you create a `StringTokenizer`, you can specify a string of delimiter characters to use for the entire string, or you can rely on the default whitespace delimiters. You can also specify whether the delimiters themselves should be returned as tokens. Finally, you can optionally specify a new string of delimiter characters when you call `nextToken()`.

Object	StringTokenizer	Enumeration
--------	-----------------	-------------

```
public class StringTokenizer implements Enumeration {
// Public Constructors
    public StringTokenizer(String str);
    public StringTokenizer(String str, String delim);
    public StringTokenizer(String str, String delim, boolean returnDelims);
// Public Instance Methods
    public int countTokens();
    public boolean hasMoreTokens();
    public String nextToken();
    public String nextToken(String delim);
// Methods Implementing Enumeration
    public boolean hasMoreElements();
    public Object nextElement();
}
```

## Timer

Java 1.3

java.util

This class implements a timer: its methods allow you to schedule one or more runnable `TimerTask` objects to be executed (once or repetitively) by a background thread at a specified time in the future. You can create a timer with the `Timer()` constructor. The no-argument version of this constructor creates a regular non-daemon background thread, which means that the Java VM will not terminate while the timer thread is running. Pass `true` to the constructor if you want the background thread to be a daemon thread.

Once you have created a `Timer`, you can schedule `TimerTask` objects to be run in the future with the various `schedule()` and `scheduleAtFixedRate()` methods. To schedule a task for a single execution, use one of the two-argument `schedule()` methods and specify the desired execution time either as a number of milliseconds in the future or as an abso-

## Timer

lute `Date`. If the number of milliseconds is 0, or if the `Date` object represents a time already passed, the task is scheduled for immediate execution.

To schedule a repeating task, use one of the three-argument versions of `schedule()` or `scheduleAtFixedRate()`. These methods are passed an argument that specifies the time (either as a number of milliseconds or as a `Date` object) of the first execution of the task and another argument, *period*, that specifies the number of milliseconds between repeated executions of the task. The `schedule()` methods schedule the task for *fixed-interval* execution. That is, each execution is scheduled for *period* milliseconds after the previous execution *ends*. Use `schedule()` for tasks such as animation, where it is important to have a relatively constant interval between executions. The `scheduleAtFixedRate()` methods, on the other hand, schedule tasks for *fixed-rate* execution. That is, each repetition of the task is scheduled for *period* milliseconds after the previous execution begins. Use `scheduleAtFixedRate()` for tasks, such as updating a clock display, that must occur at specific absolute times rather than at fixed intervals.

A single `Timer` object can comfortably schedule many `TimerTask` objects. Note, however, that all tasks scheduled by a single `Timer` share a single thread. If you are scheduling many rapidly repeating tasks, or if some tasks take a long time to execute, other tasks may have their scheduled executions delayed.

When you are done with a `Timer`, call `cancel()` to stop its associated thread from running. This is particularly important when you are using a timer whose associated thread is not a daemon thread, because otherwise the timer thread can prevent the Java VM from exiting. To cancel the execution of a particular task, use the `cancel()` method of `TimerTask`.

```
public class Timer {  
    // Public Constructors  
    public Timer();  
    public Timer(boolean isDaemon);  
    // Public Instance Methods  
    public void cancel();  
    public void schedule(TimerTask task, long delay);  
    public void schedule(TimerTask task, java.util.Date time);  
    public void schedule(TimerTask task, java.util.Date firstTime, long period);  
    public void schedule(TimerTask task, long delay, long period);  
    public void scheduleAtFixedRate(TimerTask task, java.util.Date firstTime, long period);  
    public void scheduleAtFixedRate(TimerTask task, long delay, long period);  
}
```

## TimerTask

Java 1.3

java.util

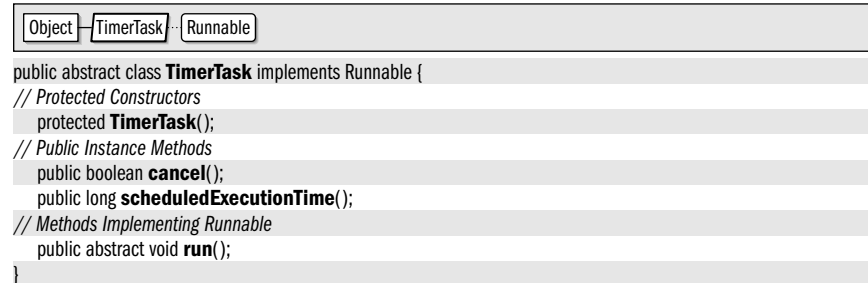
runnable

This abstract `Runnable` class represents a task that is scheduled with a `Timer` object for one-time or repeated execution in the future. You can define a task by subclassing `TimerTask` and implementing the abstract `run()` method. Schedule the task for future execution by passing an instance of your subclass to one of the `schedule()` or `scheduleAtFixedRate()` methods of `Timer`. The `Timer` object will then invoke the `run()` method at the scheduled time or times.

Call `cancel()` to cancel the one-time or repeated execution of a `TimerTask`. This method returns `true` if a pending execution was actually canceled. It returns `false` if the task has already been canceled, was never scheduled, or was scheduled for one-time execution and has already been executed. `scheduledExecutionTime()` returns the time in milliseconds at which the most recent execution of the `TimerTask` was scheduled to occur. When the host system is heavily loaded, the `run()` method may not be invoked exactly when scheduled. Some tasks may choose to do nothing if they are not invoked on time. The `run()` method can compare the return values of `scheduledExecutionTime()` and



`System.currentTimeMillis()` to determine whether the current invocation is sufficiently timely.



*Passed To:* `java.util.Timer.{schedule(), scheduleAtFixedRate()}`

## TimeZone

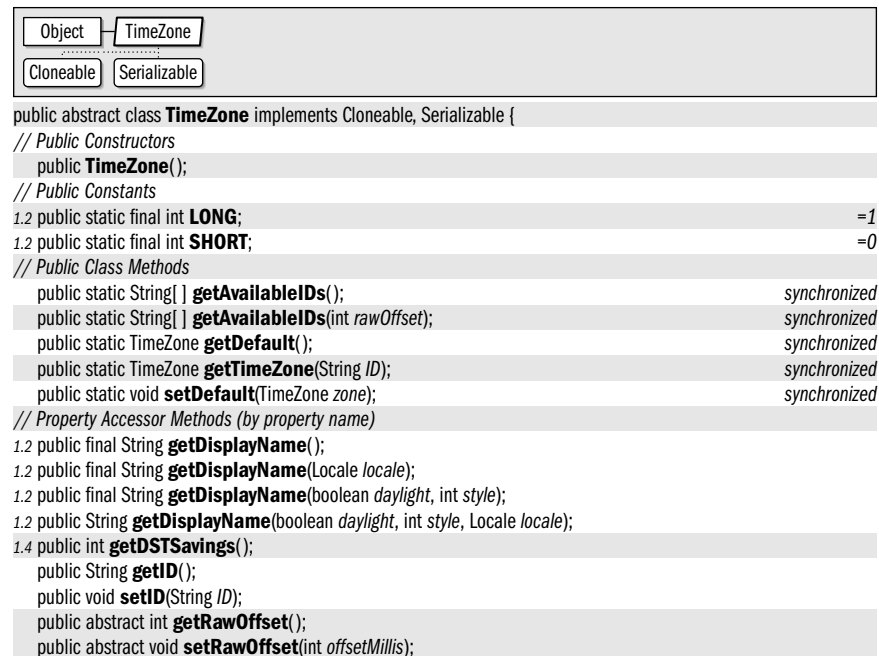
Java 1.1

`java.util`

*cloneable serializable*

The `TimeZone` class represents a time zone; it is used with the `Calendar` and `DateFormat` classes. As an abstract class, `TimeZone` cannot be directly instantiated. Instead, you should call the static `getDefault()` method to obtain a `TimeZone` object that represents the time zone inherited from the host operating system. Or you can call the static `getTimeZone()` method with the name of the desired zone. You can obtain a list of the supported time-zone names by calling the static `getAvailableIDs()` method.

Once you have a `TimeZone` object, you can call `inDaylightTime()` to determine whether, for a given `Date`, daylight-savings time is in effect for that time zone. Call `getID()` to obtain the name of the time zone. Call `getOffset()` for a given date to determine the number of milliseconds to add to GMT to convert to the time zone.



## TimeZone

```
// Public Instance Methods
1.4 public int getOffset(long date);
    public abstract int getOffset(int era, int year, int month, int day, int dayOfWeek, int milliseconds);
1.2 public boolean hasSameRules(TimeZone other);
    public abstract boolean inDaylightTime(java.util.Date date);
    public abstract boolean useDaylightTime();
// Public Methods Overriding Object
    public Object clone();
}
```

**Subclasses:** SimpleTimeZone

**Passed To:** java.text.DateFormat.setTimeZone(), Calendar.{Calendar(), getInstance(), setTimeZone()},  
GregorianCalendar.GregorianCalendar(), SimpleTimeZone.hasSameRules(), TimeZone.{hasSameRules(),  
setDefault()}

**Returned By:** java.text.DateFormat.getTimeZone(), Calendar.getTimeZone(), TimeZone.{getDefault(),  
getTimeZone()}

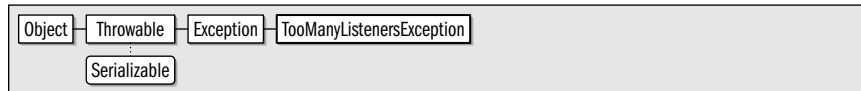
## TooManyListenersException

Java 1.1

java.util

*serializable checked*

This exception signals that an AWT component, JavaBeans component, or Swing component can have only one `EventListener` object registered for some specific type of event. That is, it signals that a particular event is a unicast event rather than a multicast event. This exception type serves a formal purpose in the Java event model; its presence in the throws clause of an `EventListener` registration method (even if the method never actually throws the exception) signals that an event is a unicast event.



```
public class TooManyListenersException extends Exception {
// Public Constructors
    public TooManyListenersException();
    public TooManyListenersException(String s);
}
```

**Thrown By:** java.awt.dnd.DragGestureRecognizer.addDragGestureListener(),  
java.awt.dnd.DragSourceContext.addDragSourceListener(),  
java.awt.dnd.DropTarget.addDropTargetListener(),  
java.beans.beancontext.BeanContextServices.getService(),  
java.beans.beancontext.BeanContextServicesSupport.getService()

## TreeMap

Java 1.2

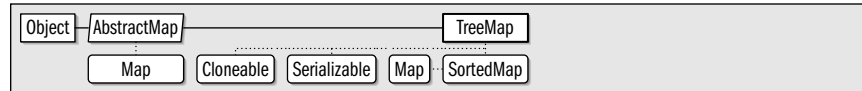
java.util

*cloneable serializable collection*

This class implements the `SortedMap` interface using an internal Red-Black tree data structure and guarantees that the keys and values of the mapping can be enumerated in ascending order of keys. `TreeMap` supports all optional `Map` methods. The objects used as keys in a `TreeMap` must all be mutually `Comparable`, or an appropriate `Comparator` must be provided when the `TreeMap` is created. Because `TreeMap` is based on a binary tree data structure, the `get()`, `put()`, `remove()`, and `containsKey()` methods operate in relatively efficient logarithmic time. If you do not need the sorting capability of `TreeMap`, however, use `HashMap` instead, as it is even more efficient. See `Map` and `SortedMap` for details on the methods of `TreeMap`. See also the related `TreeSet` class.

In order for a `TreeMap` to work correctly, the comparison method from the `Comparable` or `Comparator` interface must be consistent with the `equals()` method. That is, the `equals()` method must compare two objects as equal if and only if the comparison method also indicates those two objects are equal.

The methods of `TreeMap` are not `synchronized`. If you are working in a multithreaded environment, you must explicitly synchronize all code that modifies the `TreeMap`, or obtain a `synchronized` wrapper with `Collections.synchronizedMap()`.



```
public class TreeMap extends AbstractMap implements Cloneable, Serializable, SortedMap {
```

```
// Public Constructors
```

```
public TreeMap();
```

```
public TreeMap(Comparator c);
```

```
public TreeMap(SortedMap m);
```

```
public TreeMap(Map m);
```

```
// Methods Implementing Map
```

```
public void clear();
```

```
public boolean containsKey(Object key);
```

```
public boolean containsValue(Object value);
```

```
public Set entrySet();
```

```
public Object get(Object key);
```

```
public Set keySet();
```

```
public Object put(Object key, Object value);
```

```
public void putAll(Map map);
```

```
public Object remove(Object key);
```

```
public int size();
```

```
public Collection values();
```

```
// Methods Implementing SortedMap
```

```
public Comparator comparator();
```

```
public Object firstKey();
```

```
public SortedMap headMap(Object toKey);
```

```
public Object lastKey();
```

```
public SortedMap subMap(Object fromKey, Object toKey);
```

```
public SortedMap tailMap(Object fromKey);
```

```
// Public Methods Overriding AbstractMap
```

```
public Object clone();
```

```
}
```

## TreeSet

Java 1.2

java.util

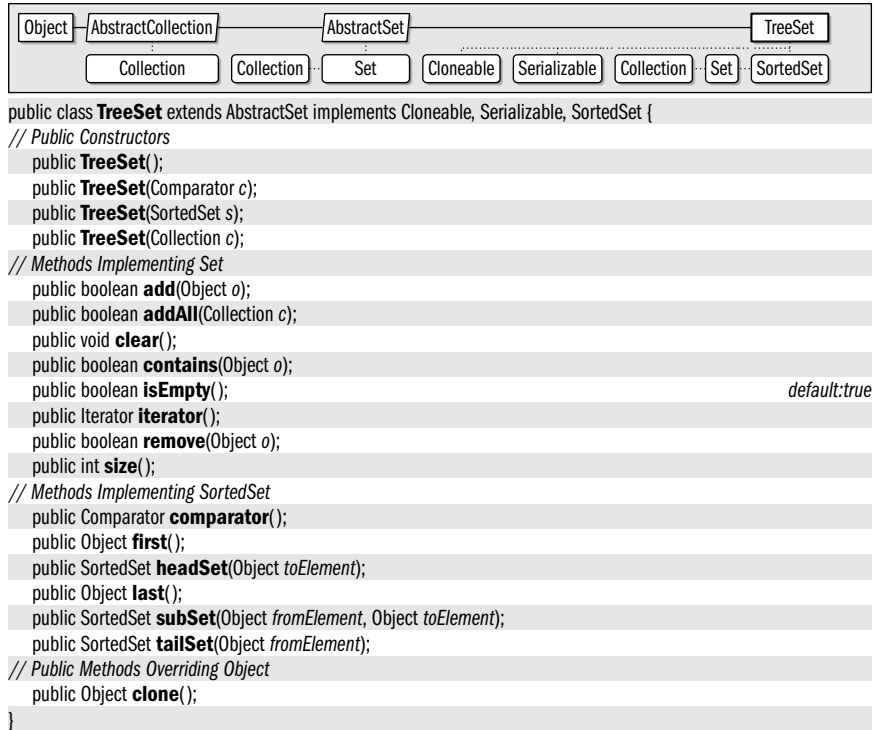
*cloneable serializable collection*

This class implements `SortedSet`, provides support for all optional methods, and guarantees that the elements of the set can be enumerated in ascending order. In order to be sorted, the elements of the set must all be mutually `Comparable` objects, or they must all be compatible with a `Comparator` object that is specified when the `TreeSet` is created. `TreeSet` is implemented on top of a `TreeMap`, so its `add()`, `remove()`, and `contains()` methods all operate in relatively efficient logarithmic time. If you do not need the sorting capability of `TreeSet`, however, use `HashSet` instead, as it is significantly more efficient. See `Set`, `SortedSet`, and `Collection` for details on the methods of `TreeSet`.

For a `TreeSet` to operate correctly, the `Comparable` or `Comparator` comparison method must be consistent with the `equals()` method. That is, the `equals()` method must compare two objects as equal if and only if the comparison method also indicates those two objects are equal.

## TreeSet

The methods of `TreeSet` are not synchronized. If you are working in a multithreaded environment, you must explicitly synchronize code that modifies the contents of the set, or obtain a synchronized wrapper with `Collections.synchronizedSet()`.



## Vector

Java 1.0

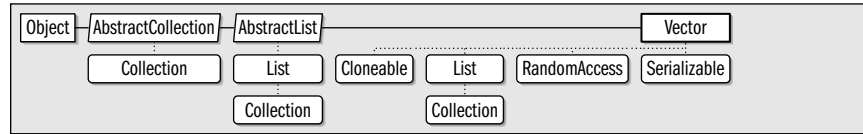
java.util

cloneable serializable collection

This class implements an ordered collection—essentially an array—of objects that can grow or shrink as necessary. `Vector` is useful when you need to keep track of a number of objects, but do not know in advance how many there will be. Use `setElementAt()` to set the object at a given index of a `Vector`. Use `elementAt()` to retrieve the object stored at a specified index. Note that you typically must cast the `Object` returned by `elementAt()` to the desired type. Call `add()` to append an object to the end of the `Vector` or to insert an object at any specified position. Use `removeElementAt()` to delete the element at a specified index or `removeElement()` to remove a specified object from the vector. `size()` returns the number of objects currently in the `Vector`. `elements()` returns an `Enumeration` that allows you to iterate through those objects. `capacity()` is not the same as `size()`; it returns the maximum number of objects a `Vector` can hold before its internal storage must be resized. `Vector` automatically resizes its internal storage for you, but if you know in advance how many objects a `Vector` will contain, you can increase its efficiency by pre-allocating this many elements with `ensureCapacity()`.

`Vector` has been part of the `java.util` package since Java 1.0, but in Java 1.2 it has been enhanced to implement the `List` interface. `List` defines new names for many of the methods already present in `Vector`; see `List` for details on those methods. `Vector` is quite similar to the `ArrayList` class, except that the methods of `Vector` are synchronized, which makes

them thread-safe but increases the overhead of calling them. If you need thread safety or need to be compatible with Java 1.0 or Java 1.1, use **Vector**; otherwise, use **ArrayList**.



```

public class Vector extends AbstractList implements Cloneable, java.util.List, RandomAccess, Serializable {
// Public Constructors
    public Vector();
    1.2 public Vector(Collection c);
    public Vector(int initialCapacity);
    public Vector(int initialCapacity, int capacityIncrement);
// Public Instance Methods
    public void addElement(Object obj); synchronized
    public int capacity(); synchronized
    public boolean contains(Object elem); Implements:List
    public void copyInto(Object[] anArray); synchronized
    public Object elementAt(int index); synchronized
    public Enumeration elements();
    public void ensureCapacity(int minCapacity); synchronized
    public Object firstElement(); synchronized
    public int indexOf(Object elem); Implements:List
    public int indexOf(Object elem, int index); synchronized
    public void insertElementAt(Object obj, int index); synchronized
    public boolean isEmpty(); Implements:List synchronized default:true
    public Object lastElement(); synchronized
    public int lastIndexOf(Object elem); Implements:List synchronized
    public int lastIndexOf(Object elem, int index); synchronized
    public void removeAllElements(); synchronized
    public boolean removeElement(Object obj); synchronized
    public void removeElementAt(int index); synchronized
    public void setElementAt(Object obj, int index); synchronized
    public void setSize(int newSize); synchronized
    public int size(); Implements:List synchronized
    public void trimToSize(); synchronized
// Methods Implementing List
    1.2 public boolean add(Object o); synchronized
    1.2 public void add(int index, Object element);
    1.2 public boolean addAll(Collection c); synchronized
    1.2 public boolean addAll(int index, Collection c); synchronized
    1.2 public void clear();
        public boolean contains(Object elem);
    1.2 public boolean containsAll(Collection c); synchronized
    1.2 public boolean equals(Object o); synchronized
    1.2 public Object get(int index); synchronized
    1.2 public int hashCode(); synchronized
        public int indexOf(Object elem);
        public boolean isEmpty(); synchronized default:true
        public int lastIndexOf(Object elem); synchronized
    1.2 public boolean remove(Object o);
    1.2 public Object remove(int index); synchronized
    1.2 public boolean removeAll(Collection c); synchronized
    1.2 public boolean retainAll(Collection c); synchronized
    1.2 public Object set(int index, Object element); synchronized
  
```

## Vector

```
public int size(); synchronized
1.2 public java.util.List subList(int fromIndex, int toIndex); synchronized
1.2 public Object[] toArray(); synchronized
1.2 public Object[] toArray(Object[] a); synchronized
// Protected Methods Overriding AbstractList
1.2 protected void removeRange(int fromIndex, int toIndex);
// Public Methods Overriding AbstractCollection
public String toString(); synchronized
// Public Methods Overriding Object
public Object clone(); synchronized
// Protected Instance Fields
protected int capacityIncrement;
protected int elementCount;
protected Object[] elementData;
}
```

*Subclasses:* Stack

*Passed To:* Too many methods to list.

*Returned By:* java.awt.image.BufferedImage.getSources(),  
java.awt.image.RenderedImage.getSources(),  
java.awt.image.renderable.ParameterBlock.{getParameters(), getSources()},  
java.awt.image.renderable.RenderableImage.getSources(),  
java.awt.image.renderable.RenderableImageOp.getSources(),  
javax.swing.plaf.basic.BasicDirectoryModel.{getDirectories(), getFiles()},  
javax.swing.table.DefaultTableModel.{convertToVector(), getDataVector()},  
javax.swing.text.GapContent.getPositionInRange(), javax.swing.text.StringContent.getPositionInRange()

*Type Of:* Too many fields to list.

## WeakHashMap

Java 1.2

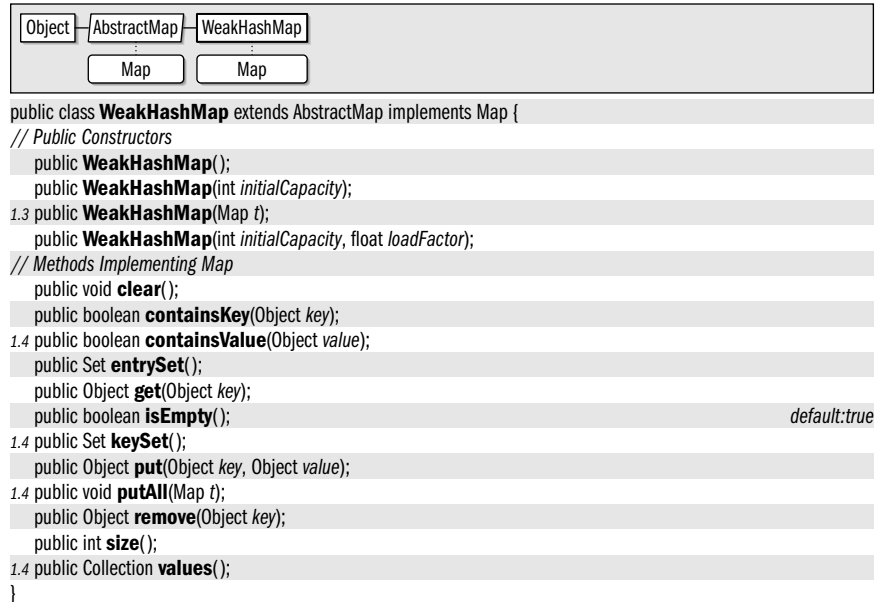
java.util

collection

This class implements **Map** using an internal hashtable. It is similar in features and performance to **HashMap**, except that it uses the capabilities of the **java.lang.ref** package, so that the key-to-value mappings it maintains do not prevent the key objects from being reclaimed by the garbage collector. When there are no more references to a key object except for the weak reference maintained by the **WeakHashMap**, the garbage collector reclaims the object, and the **WeakHashMap** deletes the mapping between the reclaimed key and its associated value. If there are no references to the value object except for the one maintained by the **WeakHashMap**, the value object also becomes available for garbage collection. Thus, you can use a **WeakHashMap** to associate an auxiliary value with an object without preventing either the object (the key) or the auxiliary value from being reclaimed. See **HashMap** for a discussion of the implementation features of this class. See **Map** for a description of the methods it defines.

**WeakHashMap** is primarily useful with objects whose **equals()** methods use the **==** operator for comparison. It is less useful with key objects of type **String**, for example, because there can be multiple **String** objects that are equal to one another and, even if the origi-

nal key value has been reclaimed by the garbage collector, it is always possible to pass a `String` with the same value to the `get()` method.



## Package java.util.jar

Java 1.2

The `java.util.jar` package contains classes for reading and writing JAR (Java ARchive) files. It is based on the `java.util.zip` package. A JAR file is nothing more than a ZIP file whose first entry is a specially named manifest file that contains attributes and digital signatures for the ZIP file entries that follow it. Many of the classes in this package are relatively simple extensions of classes from the `java.util.zip` package.

The easiest way to read a JAR file is with the random-access `JarFile` class. This class allows you to obtain the `JarEntry` that describes any named file within the JAR archive. It also allows you to obtain an enumeration of all entries in the archive and an `InputStream` for reading the bytes of a specific `JarEntry`. Each `JarEntry` describes a single entry in the archive and allows access to the `Attributes` and the digital signatures associated with the entry. The `JarFile` also provides access to the `Manifest` object for the JAR archive; this object contains `Attributes` for all entries in the JAR file. `Attributes` is a mapping of attribute name/value pairs, of course, and the inner class `Attributes.Name` defines constants for various standard attribute names.

You can also read a JAR file with `JarInputStream`. This class requires to you read each entry of the file sequentially, however. `JarOutputStream` allows you to write out a JAR file sequentially. Finally, you can also read an entry within a JAR file and manifest attributes for that entry with a `java.net.JarURLConnection` object.

### Collections:

```
public class Attributes implements Cloneable, java.util.Map;
```

*Package java.util.jar*

*Other Classes:*

```
public static class Attributes.Name;  
public class JarEntry extends java.util.zip.ZipEntry;  
public class JarFile extends java.util.zip.ZipFile;  
public class JarInputStream extends java.util.zip.ZipInputStream;  
public class JarOutputStream extends java.util.zip.ZipOutputStream;  
public class Manifest implements Cloneable;
```

*Exception:*

```
public class JarException extends java.util.zip.ZipException;
```

## Attributes

Java 1.2

java.util.jar

*cloneable collection*

This class is a `java.util.Map` that maps the attribute names of a JAR file manifest to arbitrary string values. The JAR manifest format specifies that attribute names can contain only the ASCII characters A to Z (uppercase and lowercase), the digits 0 through 9, and the hyphen and underscore characters. Thus, this class uses `Attributes.Name` as the type of attribute names, in addition to the more general `String` class. Although you can create your own `Attributes` objects, you more commonly obtain `Attributes` objects from a `Manifest`.



```
public class Attributes implements Cloneable, java.util.Map {  
    // Public Constructors  
    public Attributes();  
    public Attributes(java.util.jar.Attributes attr);  
    public Attributes(int size);  
    // Inner Classes  
    public static class Name;  
    // Public Instance Methods  
    public String getValue(String name);  
    public String getValue(Attributes.Name name);  
    public String putValue(String name, String value);  
    // Methods Implementing Map  
    public void clear();  
    public boolean containsKey(Object name);  
    public boolean containsValue(Object value);  
    public java.util.Set entrySet();  
    public boolean equals(Object o);  
    public Object get(Object name);  
    public int hashCode();  
    public boolean isEmpty(); default:true  
    public java.util.Set keySet();  
    public Object put(Object name, Object value);  
    public void putAll(java.util.Map attr);  
    public Object remove(Object name);  
    public int size();  
    public java.util.Collection values();  
    // Public Methods Overriding Object  
    public Object clone();  
}
```



```
// Protected Instance Fields
protected java.util.Map map;
}
```

*Passed To:* java.util.jar.Attributes.Attributes()

*Returned By:* java.net.JarURLConnection.{getAttributes(), getMainAttributes()},  
JarEntry.getAttributes(), Manifest.{getAttributes(), getMainAttributes()}

## Attributes.Name

Java 1.2

java.util.jar

This class represents the name of an attribute in an `Attributes` object. It defines constants for the various standard attribute names used in JAR file manifests. Attribute names can contain only ASCII letters, digits, and the hyphen and underscore characters. Any other Unicode characters are illegal.

```
public static class Attributes.Name {
// Public Constructors
    public Name(String name);
// Public Constants
    public static final Attributes.Name CLASS_PATH;
    public static final Attributes.Name CONTENT_TYPE;
    1.3 public static final Attributes.Name EXTENSION_INSTALLATION;
    1.3 public static final Attributes.Name EXTENSION_LIST;
    1.3 public static final Attributes.Name EXTENSION_NAME;
    public static final Attributes.Name IMPLEMENTATION_TITLE;
    1.3 public static final Attributes.Name IMPLEMENTATION_URL;
    public static final Attributes.Name IMPLEMENTATION_VENDOR;
    1.3 public static final Attributes.Name IMPLEMENTATION_VENDOR_ID;
    public static final Attributes.Name IMPLEMENTATION_VERSION;
    public static final Attributes.Name MAIN_CLASS;
    public static final Attributes.Name MANIFEST_VERSION;
    public static final Attributes.Name SEALED;
    public static final Attributes.Name SIGNATURE_VERSION;
    public static final Attributes.Name SPECIFICATION_TITLE;
    public static final Attributes.Name SPECIFICATION_VENDOR;
    public static final Attributes.Name SPECIFICATION_VERSION;
// Public Methods Overriding Object
    public boolean equals(Object o);
    public int hashCode();
    public String toString();
}
```

*Passed To:* java.util.jar.Attributes.getValue()

*Type Of:* Too many fields to list.

## JarEntry

Java 1.2

java.util.jar

cloneable

This class extends `java.util.zip.ZipEntry`; it represents a single file in a JAR archive and the manifest attributes and digital signatures associated with that file. `JarEntry` objects can be

## *JarEntry*

read from a JAR file with `JarFile` or `JarInputStream`, and they can be written to a JAR file with `JarOutputStream`. Use `getAttributes()` to obtain the `Attributes` for the entry. Use `getCertificates()` to obtain a `java.security.cert.Certificate` array that contains the certificate chains for all digital signatures associated with the file.



```
public class JarEntry extends java.util.zip.ZipEntry {
// Public Constructors
    public JarEntry(JarEntry je);
    public JarEntry(String name);
    public JarEntry(java.util.zip.ZipEntry ze);
// Public Instance Methods
    public java.util.jar.Attributes getAttributes() throws java.io.IOException;
    public java.security.cert.Certificate[ ] getCertificates();
}
```

*Passed To:* `JarEntry.JarEntry()`

*Returned By:* `java.net.JarURLConnection.getJarEntry()`, `JarFile.getJarEntry()`, `JarInputStream.getNextJarEntry()`

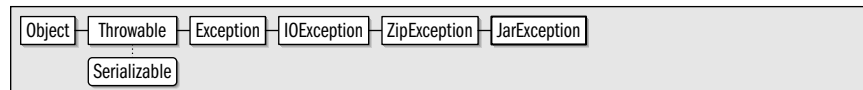
## **JarException**

**Java 1.2**

`java.util.jar`

*serializable checked*

This exception signals an error while reading or writing a JAR file.



```
public class JarException extends java.util.zip.ZipException {
// Public Constructors
    public JarException();
    public JarException(String s);
}
```

## **JarFile**

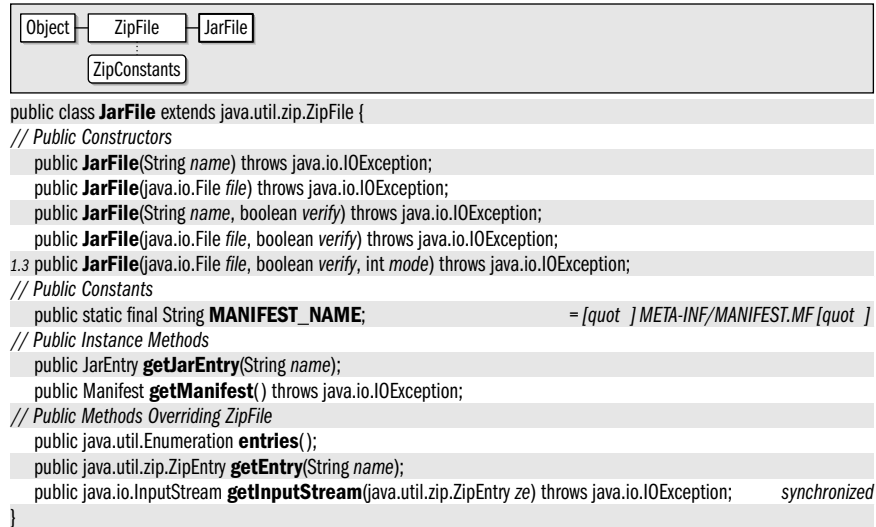
**Java 1.2**

`java.util.jar`

This class represents a JAR file and allows the manifest, file list, and individual files to be read from the JAR file. It extends `java.util.zip.ZipFile`, and its use is similar to that of its superclass. Create a `JarFile` by specifying a filename or `File` object. If you do not want `JarFile` to attempt to verify any digital signatures contained in the `JarFile`, pass an optional boolean argument of `false` to the `JarFile()` constructor. In Java 1.3, temporary JAR files can be automatically deleted when they are closed. To take advantage of this feature, pass `ZipFile.OPEN_READ|ZipFile.OPEN_DELETE` as the *mode* argument to the `JarFile()` constructor.

Once you have created a `JarFile` object, obtain the JAR Manifest with `getManifest()`. Obtain an enumeration of the `java.util.zip.ZipEntry` objects in the file with `entries()`. Get the `JarEntry` for a specified file in the JAR file with `getJarEntry()`. To read the contents of a specific entry in the JAR file, obtain the `JarEntry` or `ZipEntry` object that represents that entry, pass

it to `getInputStream()`, and then read until the end of that stream. `JarFile` does not support the creation of new JAR files or the modification of existing files.



*Returned By:* `java.net.JarURLConnection.getJarFile()`

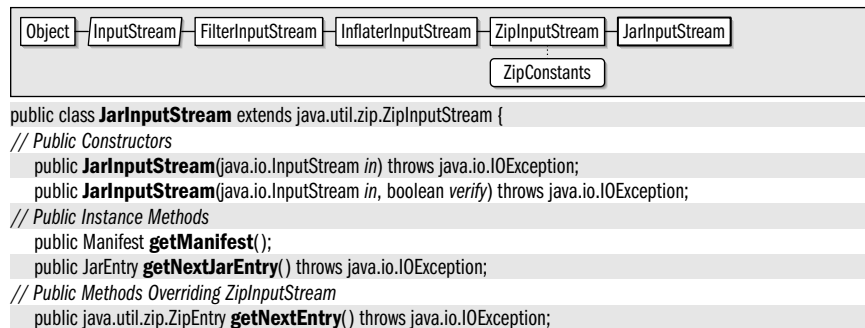
## JarInputStream

Java 1.2

`java.util.jar`

This class allows a JAR file to be read from an input stream. It extends `java.util.ZipInputStream` and is used much like that class is used. To create a `JarInputStream`, simply specify the `InputStream` from which to read. If you do not want the `JarInputStream` to attempt to verify any digital signatures contained in the JAR file, pass `false` as the second argument to the `JarInputStream()` constructor. The `JarInputStream()` constructor first reads the JAR manifest entry, if one exists. The manifest must be the first entry in the JAR file. `getManifest()` returns the `Manifest` object for the JAR file.

Once you have created a `JarInputStream`, call `getNextJarEntry()` or `getNextEntry()` to obtain the `JarEntry` or `java.util.zip.ZipEntry` object that describes the next entry in the JAR file. Then, call a `read()` method (including the inherited versions) to read the contents of that entry. When the stream reaches the end of file, call `getNextJarEntry()` again to start reading the next entry in the file. When all entries have been read from the JAR file, `getNextJarEntry()` and `getNextEntry()` return null.



## *JarInputStream*

```
public int read(byte[] b, int off, int len) throws java.io.IOException;
// Protected Methods Overriding ZipInputStream
protected java.util.zip.ZipEntry createZipEntry(String name);
}
```

## **JarOutputStream**

**Java 1.2**

**java.util.jar**

This class can write a JAR file to an arbitrary `OutputStream`. `JarOutputStream` extends `java.util.zip.ZipOutputStream` and is used much like that class is used. Create a `JarOutputStream` by specifying the stream to write to and, optionally, the `Manifest` object for the JAR file. The `JarOutputStream()` constructor starts by writing the contents of the `Manifest` object into an appropriate JAR file entry. It is the programmer's responsibility to ensure that the contents of the JAR entries written subsequently match those specified in the `Manifest` object. This class provides no explicit support for attaching digital signatures to entries in the JAR file.

After creating a `JarOutputStream`, call `putNextEntry()` to specify the `JarEntry` or `java.util.zip.ZipEntry` to be written to the stream. Then, call any of the inherited `write()` methods to write the contents of the entry to the stream. When that entry is finished, call `putNextEntry()` again to begin writing the next entry. When you have written all JAR file entries in this way, call `close()`. Before writing any entry, you may call the inherited `setMethod()` and `setLevel()` methods to specify how the entry should be compressed. See `java.util.zip.ZipOutputStream`.



```
public class JarOutputStream extends java.util.zip.ZipOutputStream {
// Public Constructors
    public JarOutputStream(java.io.OutputStream out) throws java.io.IOException;
    public JarOutputStream(java.io.OutputStream out, Manifest man) throws java.io.IOException;
// Public Methods Overriding ZipOutputStream
    public void putNextEntry(java.util.zip.ZipEntry ze) throws java.io.IOException;
}
```

## **Manifest**

**Java 1.2**

**java.util.jar**

**cloneable**

This class represents the manifest entry of a JAR file. `getMainAttributes()` returns an `Attributes` object that represents the manifest attributes that apply to the entire JAR file. `getAttributes()` returns an `Attributes` object that represents the manifest attributes specified for a single file in the JAR file. `getEntries()` returns a `java.util.Map` that maps the names of entries in the JAR file to the `Attributes` objects associated with those entries. `getEntries()` returns the `Map` object used internally by the `Manifest`. You can edit the contents of the `Manifest` by adding, deleting, or editing entries in the `Map`. `read()` reads manifest entries from an input stream, merging them into the current set of entries. `write()` writes the `Manifest` out to the specified output stream.



```
public class Manifest implements Cloneable {
// Public Constructors
```

```

public Manifest();
public Manifest(Manifest man);
public Manifest(java.io.InputStream is) throws java.io.IOException;
// Public Instance Methods
public void clear();
public java.util.jar.Attributes getAttributes(String name);
public java.util.Map getEntries();                                default:HashMap
public java.util.jar.Attributes getMainAttributes();
public void read(java.io.InputStream is) throws java.io.IOException;
public void write(java.io.OutputStream out) throws java.io.IOException;
// Public Methods Overriding Object
public Object clone();
public boolean equals(Object o);
public int hashCode();
}

```

**Passed To:** `java.net.URLClassLoader.definePackage()`, `JarOutputStream.JarOutputStream()`, `Manifest.Manifest()`

**Returned By:** `java.net.JarURLConnection.getManifest()`, `JarFile.getManifest()`, `JarInputStream.getManifest()`

## Package *java.util.logging*

**Java 1.4**

The `java.util.logging` package defines a sophisticated and highly configurable logging facility that Java applications can use to emit, filter, format, and output warning, diagnostic, tracing, and debugging messages. An application generates log messages by calling various methods of a `Logger` object. The content of a log message (with other pertinent details such as the time and sequence number) is encapsulated in a `LogRecord` object generated by the `Logger`. A `Handler` object represents a destination for `LogRecord` objects. Concrete subclasses of `Handler` support destinations such as files and sockets. Most `Handler` objects have an associated `Formatter` that converts a `LogRecord` object into the actual text that is logged. The subclasses `SimpleFormatter` and `XMLFormatter` produce simple plain-text log messages and detailed XML logs respectively.

Each log message has an associated severity level. The `Level` class defines a type-safe enumeration of defined levels. `Logger` and `Handler` objects both have an associated `Level`, and discard any log messages whose severity is less than that specified level. In addition to this level-based filtering, `Logger` and `Handler` objects may also have an associated `Filter` object which may be implemented to filter log messages based on any desired criteria.

Applications that desire complete control over the logs they generate can create a `Logger` object, along with `Handler`, `Formatter` and `Filter` objects that control the destination, content, and appearance of the log. Simpler applications need only to create a `Logger` for themselves, and can leave the rest to the `LogManager` class. `LogManager` reads a system-wide configuration file (or a configuration class) and automatically directs log messages to a standard destination (or destinations) for the system.

### *Interfaces:*

public interface **Filter**;

*Package java.util.logging*

*Classes:*

```
public class ErrorManager;
public abstract class Formatter;
    public class SimpleFormatter extends Formatter;
    public class XMLFormatter extends Formatter;
public abstract class Handler;
    public class MemoryHandler extends Handler;
    public class StreamHandler extends Handler;
        public class ConsoleHandler extends StreamHandler;
        public class FileHandler extends StreamHandler;
        public class SocketHandler extends StreamHandler;
public class Level implements Serializable;
public class Logger;
public final class LoggingPermission extends java.security.BasicPermission;
public class LogManager;
public class LogRecord implements Serializable;
```

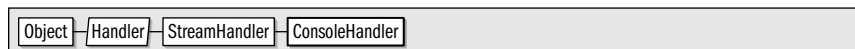
## ConsoleHandler

Java 1.4

java.util.logging

This Handler subclass formats LogRecord objects and outputs the resulting string to the System.err output stream. When a ConsoleHandler is created, the various properties inherited from Handler are initialized using system-wide defaults obtained by querying named values with LogManager.getProperty(). The following table lists these properties: the value passed to getProperty() and the default value used if getProperty() returns null. See Handler for further details.

Handler property	LogManager property name	Default
level	java.util.logging.ConsoleHandler.level	Level.INFO
filter	java.util.logging.ConsoleHandler.filter	null
formatter	java.util.logging.ConsoleHandler.formatter	SimpleFormatter
encoding	java.util.logging.ConsoleHandler.encoding	platform default



```
public class ConsoleHandler extends StreamHandler {
    // Public Constructors
    public ConsoleHandler();
    // Public Methods Overriding StreamHandler
    public void close();
    public void publish(LogRecord record);
}
```

## ErrorManager

Java 1.4

java.util.logging

An important feature of the Logging API is that the logging methods called by applications never throw exceptions: it is not reasonable to expect programmers to nest all their logging calls within try/catch blocks, and even if they did, there is no useful way for an application to recover from an exception in the logging subsystem. Since handler classes such as FileHandler are inherently subject to I/O exceptions, the ErrorManager provides a way for a handler to report an exception instead of simply discarding it.

All Handler objects have an instance of `ErrorManager` associated with them. If an exception occurs in the handler, it passes the exception, along with a message and one of the error code constants defined by `ErrorManager` to the `error()` method. `error()` writes a message describing the exception to `System.err`, but does so only the first time it is called: the expectation is that a Handler that throws an exception once will continue to throw the same exception with each subsequent log message, and it is not useful to flood `System.err` with repeated error messages. You can of course define subclasses of `ErrorManager` that override `error()` to provide some other reporting mechanism. If you do this, register an instance of your custom `ErrorManager` by calling the `setErrorHandler()` method of your Handler.

```
public class ErrorManager {
// Public Constructors
    public ErrorManager();
// Public Constants
    public static final int CLOSE_FAILURE;           =3
    public static final int FLUSH_FAILURE;           =2
    public static final int FORMAT_FAILURE;          =5
    public static final int GENERIC_FAILURE;         =0
    public static final int OPEN_FAILURE;            =4
    public static final int WRITE_FAILURE;           =1
// Public Instance Methods
    public void error(String msg, Exception ex, int code);    synchronized
}
```

*Passed To:* `Handler.setErrorHandler()`

*Returned By:* `Handler.getErrorHandler()`

## **FileHandler**

**Java 1.4**

`java.util.logging`

This Handler subclass formats `LogRecord` objects and outputs the resulting strings to a file or to a rotating set of files. Arguments passed to the `FileHandler()` constructor specify which file or files are used, and how they are used. The arguments are optional, and if they are not specified, defaults are obtained through `LogManager.getProperty()`, as described later. The constructor arguments are:

### *pattern*

A string containing substitution characters that describes one or more files to use. The substitutions performed to convert this pattern to a filename are described below.

### *limit*

An approximate maximum file size for the log file, or 0 for no limit. If *count* is set to greater than one, then when a log file reaches this maximum, `FileHandler` closes it, renames it, and then starts a new log with the original filename.

### *count*

When *limit* is set to be nonzero, this argument specifies the number of old log files to retain.

### *append*

true if the `FileHandler` should append to log messages already in the named file, or false if it should overwrite the file.

The *pattern* argument is the most important of these; it specifies which file or files the `FileHandler` will write to. `FileHandler` performs the following substitutions on the specified pattern to convert it to a filename.

## FileHandler

For	Substitute
/	The directory separator character for the platform. This means that you can always use a forward slash in your patterns, even on Windows filesystems that use backward slashes.
%%	A single literal percent sign.
%h	The user's home directory: the value of the system property "user.home".
%t	The temporary directory for the system.
%u	A unique number to be used to distinguish this log file from other log files with the same pattern (this may be necessary when multiple Java programs are creating logs at the same time).
%g	The "generation number" of old log files when the <i>limit</i> argument is nonzero and the <i>count</i> argument is greater than 1. <i>FileHandler</i> always writes log records into a file in which %g is replaced by 0. But when that file fills up, it is closed and renamed with the 0 replaced by a 1. Older files are similarly renamed, with their generation number incremented. When the number of log files reaches the number specified by <i>count</i> , then the oldest file is deleted to make room for the new one.

When a *FileHandler* is created, the *LogManager.getProperty()* method is used to obtain defaults for any unspecified constructor arguments, and also to obtain initial values for the various properties inherited from *Handler*. The table below lists these arguments and properties, the value passed to *getProperty()*, and the default value used if *getProperty()* returns null. See *Handler* for further details.

Property or argument	LogManager property name	Default
level	java.util.logging.FileHandler.level	Level.ALL
filter	java.util.logging.FileHandler.filter	null
formatter	java.util.logging.FileHandler.formatter	XMLFormatter
encoding	java.util.logging.FileHandler.encoding	platform default
pattern	java.util.logging.FileHandler.pattern	%h/java%u.log
limit	java.util.logging.FileHandler.limit	0 (no limit)
count	java.util.logging.FileHandler.count	1
append	java.util.logging.FileHandler.append	false

```

Object -> Handler -> StreamHandler -> FileHandler

public class FileHandler extends StreamHandler {
// Public Constructors
    public FileHandler() throws java.io.IOException, SecurityException;
    public FileHandler(String pattern) throws java.io.IOException, SecurityException;
    public FileHandler(String pattern, boolean append) throws java.io.IOException, SecurityException;
    public FileHandler(String pattern, int limit, int count) throws java.io.IOException, SecurityException;
    public FileHandler(String pattern, int limit, int count, boolean append) throws java.io.IOException,
        SecurityException;
// Public Methods Overriding StreamHandler
    public void close() throws SecurityException;                                synchronized
    public void publish(LogRecord record);                                       synchronized
}

```



**Filter****Java 1.4****java.util.logging**

This interface defines the method that a class must implement if it wants to filter log messages for a `Logger` or `Handler` class. `isLoggable()` should return `true` if the specified `LogRecord` contains information that should be logged. It should return `false` if the `LogRecord` should be filtered out not appear in any destination log. Note that both `Logger` and `Handler` provide built-in filtering based on the severity level of the `LogRecord`. This `Filter` interface exists to provide a customized filtering capability.

```
public interface Filter {
    // Public Instance Methods
    public abstract boolean isLoggable(LogRecord record);
}
```

*Passed To:* `Handler.setFilter()`, `Logger.setFilter()`

*Returned By:* `Handler.getFilter()`, `Logger.getFilter()`

**Formatter****Java 1.4****java.util.logging**

A `Formatter` object is used by a `Handler` to convert a `LogRecord` to a `String` prior to logging it. Most applications can simply use one of the predefined concrete subclasses: `SimpleFormatter` or `XMLFormatter`. Applications requiring custom formatting of log messages will need to subclass this class and define the `format()` method to perform the desired conversion. Such subclasses may find the `formatMessage()` method useful; it performs localization using `java.util.ResourceBundle` and formatting using the facilities of the `java.text` package. `getHead()` and `getTail()` return a prefix and suffix (such as opening and closing XML tags) for a log file.

```
public abstract class Formatter {
    // Protected Constructors
    protected Formatter();
    // Public Instance Methods
    public abstract String format(LogRecord record);
    public String formatMessage(LogRecord record);
    public String getHead(Handler h);
    public String getTail(Handler h);
}
```

*Subclasses:* `SimpleFormatter`, `XMLFormatter`

*Passed To:* `Handler.setFormatter()`, `StreamHandler.StreamHandler()`

*Returned By:* `Handler.getFormatter()`

**Handler****Java 1.4****java.util.logging**

A `Handler` takes `LogRecord` objects from a `Logger` and, if their severity level is high enough, formats and publishes them to some destination (a file or socket, for example). The subclasses of this abstract class support various destinations, and implement destination-specific `publish()`, `flush()` and `close()` methods.

In addition to the destination-specific abstract methods, this class also defines concrete methods used by most `Handler` subclasses. These are property getter and setter methods to specify the severity `Level` of logging messages to be handled, an optional `Filter`, a `Formatter` to convert log messages from `LogRecord` objects to text, a text encoding for the output text, and an `ErrorManager` to handle any exceptions that arise during log output.

## Handler

Subclass-specific defaults for each of these properties are typically defined as properties of `LogManager` and are read from a system-wide logging configuration file.

In the simplest uses of the Logging API, a `Logger` sends its log messages to one or more handlers defined by the `LogManager` class for its “root logger”. In this case there is no need for the application to ever instantiate or use a `Handler` directly. Applications that want custom control over the destination of their logs create and configure an instance of a `Handler` subclass, but never need to call its `publish()`, `flush()` or `close()` methods directly: that is done by the `Logger`.

```
public abstract class Handler {
// Protected Constructors
    protected Handler();
// Property Accessor Methods (by property name)
    public String getEncoding();
    public void setEncoding(String encoding) throws SecurityException, java.io.UnsupportedEncodingException;
    public ErrorManager getErrorManager();
    public void setErrorManager(ErrorManager em);
    public Filter getFilter();
    public void setFilter(Filter newFilter) throws SecurityException;
    public Formatter getFormatter();
    public void setFormatter(Formatter newFormatter) throws SecurityException;
    public Level getLevel();
    public void setLevel(Level newLevel) throws SecurityException;
// Public Instance Methods
    public abstract void close() throws SecurityException;
    public abstract void flush();
    public boolean isLoggable(LogRecord record);
    public abstract void publish(LogRecord record);
// Protected Instance Methods
    protected void reportError(String msg, Exception ex, int code);
}
```

**Subclasses:** `MemoryHandler`, `StreamHandler`

**Passed To:** `Formatter.{getHead(), getTail()}`, `Logger.{addHandler(), removeHandler()}`,  
`MemoryHandler.MemoryHandler()`, `XMLFormatter.{getHead(), getTail()}`

**Returned By:** `Logger.getHandlers()`

## Level

Java 1.4

`java.util.logging`

serializable

This class defines constants that represent the seven standard severity levels for log messages plus constants that turn logging off and enable logging at any level. When logging is enabled at one severity level, it is also enabled at all higher levels. The seven level constants, in order from most severe to least severe are: `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINE`, and `FINEST`. The constant `ALL` enable logging of any message, regardless of its level. The constant `OFF` disables logging entirely. Note that these constants are all `Level` objects, rather than integers. This provides type safety.

Application code should rarely, if ever, need to use any of the methods of this class. Instead, they can simply use the constants it defines.

Object — Level — Serializable

```

public class Level implements Serializable {
// Protected Constructors
    protected Level(String name, int value);
    protected Level(String name, int value, String resourceName);
// Public Constants
    public static final Level ALL;
    public static final Level CONFIG;
    public static final Level FINE;
    public static final Level FINER;
    public static final Level FINEST;
    public static final Level INFO;
    public static final Level OFF;
    public static final Level SEVERE;
    public static final Level WARNING;
// Public Class Methods
    public static Level parse(String name) throws IllegalArgumentException;           synchronized
// Public Instance Methods
    public String getLocalizedName();
    public String getName();
    public String getResourceBundleName();
    public final int intValue();
// Public Methods Overriding Object
    public boolean equals(Object ox);
    public int hashCode();
    public final String toString();
}

```

*Passed To:* Too many methods to list.

*Returned By:* Handler.getLevel(), Level.parse(), Logger.getLevel(), LogRecord.getLevel(), MemoryHandler.getPushLevel()

*Type Of:* Level.{ALL, CONFIG, FINE, FINER, FINEST, INFO, OFF, SEVERE, WARNING}

## Logger

Java 1.4

java.util.logging

A **Logger** object is used to emit log messages. **Logger** does not have a public constructor, but there are several ways to obtain a **Logger** object to use in your code:

- Typically, applications call the static **getLogger()** method to create or lookup a named **Logger** within a hierarchy of named loggers. Loggers have dot-separated hierarchical names, which should be based on the name of the class or package that uses them. Loggers obtained in this way inherit their logging level, resource bundle (for localization) and **Handler** objects from their ancestors in the hierarchy and ultimately from the root **Logger** defined by the global **LogManager**.
- Applets that require a **Logger** with no security restrictions should use the static **getAnonymousLogger()** method to create an unnamed **Logger** that is not part of the hierarchy of named **Logger** objects managed by the **LogManager**. A **Logger** created by this method has the **LogManager** root logger as its parent, and inherits the logging level and handlers of that root logger.
- Finally, the static **Logger.global** field refers to a predefined **Logger** named "global"; programmers may find this predefined **Logger** convenient during the early stages of application development, but it should not be used in production code.

## Logger

Once a suitable `Logger` has been obtained, there are a variety of methods that can be used to create a log message:

- The `log()` methods log a specified message at the specified level, with optional parameters that can be used in message localization. These methods examine the call stack and make an attempt to determine the class and method name from which the method is emitted. Because of code optimization and just-in-time compilation techniques, however, they may not always be able to determine this information.
- The `logp()` (“log precise”) methods are like the `log()` methods but allow you to explicitly specify the name of the class and method that are emitting the log message.
- The `logrb()` methods are like the `logp()` methods, but additionally take the name of a resource bundle to use for localizing the message.
- `entering()`, `exiting()`, and `throwing()` are convenience methods for emitting log messages that trace the execution of a program. These methods use a logging level of `Level.FINER`. Note that there are variants of `entering()` and `exiting()` that allow specification of method arguments and return values.
- Finally, `Logger` defines a set of easy-to-use convenience methods for logging a simple message at a specific logging level. These methods have the same names as the logging levels: `severe()`, `warning()`, `info()`, `config()`, `fine()`, `finer()`, and `finest()`.

A `Logger` has an associated logging `Level`, and discards any log messages with a severity lower than this. The severity level is initialized from the system configuration file, which is usually the desired behavior. You can explicitly override this setting with `setLevel()`. You might want to do this if you created the `Logger` with `getAnonymousLogger()` and have read the desired logging level from a configuration file of your own. If level-based filtering of log messages is not sufficient, you can associate a `Filter` with your `Logger` by calling `setFilter`. If you do this, any log messages rejected by the `Filter` will be discarded.

A `Logger` sends its log messages to any `Handler` objects that have been registered with `addHandler()`. Call `getHandlers()` to obtain an array of all registered handlers, and call `removeHandler()` to de-register a handler. By default, all log messages are also sent to the handlers of the parent logger and any other ancestor loggers. Since all named and anonymous loggers have the `LogManager` root logger as a parent or ancestor, all loggers by default send their log messages to the handlers defined in the system logging configuration file. See `LogManager` for details. If you do not want a `Logger` to use the handlers of its ancestors, pass `false` to `setUseParentHandlers()`.

`getLogger()` and `getAnonymousLogger()` allow you to specify the name of a `java.util.ResourceBundle` for use in localizing log messages, and `logrb()` allows you to specify the name of a resource bundle to use to localize a specific log message. If a resource bundle is specified for the `Logger` or for a specific log message, then the message argument to the various logging methods is treated not as a literal message but instead as a localization key for which a localized version is to be looked up in the resource bundle. As part of the localization, any parameters, such as those specified by the `param1` and `params` arguments to the `log()` method are substituted into the localized message string as per `java.text.MessageFormat`. (Note, however that this localization and formatting is not performed by the `Logger` itself: instead, it simply stores the `ResourceBundle` and parameters in the `LogRecord`. It is the `Formatter` associated with the output `Handler` object that actually performs the localization.)

All the methods of this class are thread-safe and do not require external synchronization.

```

public class Logger {
    // Protected Constructors
    protected Logger(String name, String resourceBundleName);
    // Public Constants
    public static final Logger global;
    // Public Class Methods
    public static Logger getAnonymousLogger(); synchronized
    public static Logger getAnonymousLogger(String resourceBundleName); synchronized
    public static Logger getLogger(String name); synchronized
    public static Logger getLogger(String name, String resourceBundleName); synchronized
    // Property Accessor Methods (by property name)
    public Filter getFilter();
    public void setFilter(Filter newFilter) throws SecurityException;
    public Handler[] getHandlers(); synchronized
    public Level getLevel();
    public void setLevel(Level newLevel) throws SecurityException;
    public String getName();
    public Logger getParent();
    public void setParent(Logger parent);
    public java.util.ResourceBundle getResourceBundle();
    public String getResourceBundleName();
    public boolean getUseParentHandlers(); synchronized
    public void setUseParentHandlers(boolean useParentHandlers); synchronized
    // Public Instance Methods
    public void addHandler(Handler handler) throws SecurityException; synchronized
    public void config(String msg);
    public void entering(String sourceClass, String sourceMethod);
    public void entering(String sourceClass, String sourceMethod, Object param1);
    public void entering(String sourceClass, String sourceMethod, Object[] params);
    public void exiting(String sourceClass, String sourceMethod);
    public void exiting(String sourceClass, String sourceMethod, Object result);
    public void fine(String msg);
    public void finer(String msg);
    public void finest(String msg);
    public void info(String msg);
    public boolean isLoggable(Level level);
    public void log(LogRecord record);
    public void log(Level level, String msg);
    public void log(Level level, String msg, Throwable thrown);
    public void log(Level level, String msg, Object param1);
    public void log(Level level, String msg, Object[] params);
    public void logp(Level level, String sourceClass, String sourceMethod, String msg);
    public void logp(Level level, String sourceClass, String sourceMethod, String msg, Object param1);
    public void logp(Level level, String sourceClass, String sourceMethod, String msg, Object[] params);
    public void logp(Level level, String sourceClass, String sourceMethod, String msg, Throwable thrown);
    public void logrb(Level level, String sourceClass, String sourceMethod, String bundleName, String msg);
    public void logrb(Level level, String sourceClass, String sourceMethod, String bundleName, String msg,
        Object param1);
    public void logrb(Level level, String sourceClass, String sourceMethod, String bundleName, String msg,
        Throwable thrown);
    public void logrb(Level level, String sourceClass, String sourceMethod, String bundleName, String msg,
        Object[] params);
    public void removeHandler(Handler handler) throws SecurityException; synchronized
    public void severe(String msg);
    public void throwing(String sourceClass, String sourceMethod, Throwable thrown);

```

## Logger

```
public void warning(String msg);  
}
```

*Passed To:* `Logger.setParent()`, `LogManager.addLogger()`

*Returned By:* `Logger.getAnonymousLogger()`, `getLogger()`, `getParent()`, `LogManager.getLogger()`

*Type Of:* `Logger.global`

## LoggingPermission

Java 1.4

`java.util.logging`

*serializable permission*

This class is a `java.security.Permission` that governs the use of security-sensitive logging methods. The single defined name (or target) for `LoggingPermission` is “control”, which represents permission to invoke various logging control methods such as `Logger.setLevel()` and `LogManager.readConfiguration()`. The methods in this package that throw `SecurityException` all require a `LoggingPermission` named “control” in order to run. Application programmers never need to use this class. System administrators configuring security policies may need to be familiar with it.



```
public final class LoggingPermission extends java.security.BasicPermission {  
    // Public Constructors  
    public LoggingPermission(String name, String actions) throws IllegalArgumentException;  
}
```

## LogManager

Java 1.4

`java.util.logging`

As its name implies, this class is the manager for the `java.util.logging` API. It has three specific purposes:

1. To read a logging configuration file and create the default `Handler` objects specified in that file
2. To manage a set of `Logger` objects, arranging them into a tree based on their hierarchical names
3. To create and manage the unnamed `Logger` object that serves as the parent or ancestor of every other `Logger`

This class handles the important behind-the-scenes details that makes the Logging API work. Typical applications can make use of logging without ever having to use this class explicitly. Although its API is not commonly used by application programmers, it is still useful to understand the `LogManager` class, so it is described in detail here.

There is a single global instance of `LogManager`, which is obtained with the static `getLogManager()` method. By default, this global log manager object is an instance of the `LogManager` class itself. You may instead instantiate an instance of a subclass of `LogManager` by specifying the full class name of the subclass as the value of the system property `java.util.logging.manager`.

One of the primary purposes of the `LogManager` class is to read a `java.util.Properties` file that specifies the default logging configuration for the system. By default, this file is named `logging.properties` and is stored in the `jre/lib` directory of the Java installation. If you want to run a Java application using a different logging configuration, you can edit

the default configuration file, but it is typically easier to create a new configuration file and tell the JVM about it by setting the system property `java.util.logging.config.file` to the name of your customized configuration file.

The most important purpose of the configuration file is to specify a set of `Handler` objects to which all log messages are sent. This is done by setting the `handlers` property in the file to a space-separated list of `Handler` class names. The `LogManager` will load the specified classes and instantiate each one using the default no-arg constructor, then register those `Handler` objects on the root `Logger`, where they are inherited by all other loggers. (You'll learn more about the root logger later.) Each of these `Handler` objects further configures itself by reading additional properties from the configuration file, as described in the documentation for each handler class.

The configuration file may also contain property name that are formed by appending “.level” to the name of a logger. The value of any such property is taken as the name of a logging `Level` for the named `Logger`. When the named logger is created and registered with the `LogManager` (described below) its logging level is automatically set to the specified level.

An application or any custom `Handler` or `Formatter` subclass or `Filter` implementation can read its own properties from the logging configuration file with the `getProperty()` method of `LogManager`. This is a useful way to provide customizability for logging-related classes.

In addition to managing the configuration file properties, a second purpose of `LogManager` is to maintain a tree of `Logger` objects organized into a hierarchy based on their dot-separated hierarchical names. The `addLogger()` method registers a new `Logger` object with the `LogManager` and inserts it into the tree. This method is called automatically by the `Logger.getLogger()` factory method, however, so you never need to call it yourself. The `getLogger()` method of `LogManager` finds and returns a named `Logger` object within the tree. Use `getLoggerNames()` to obtain an `Enumeration` of the names of all registered loggers.

At the root of the tree is a root logger, created by the `LogManager`, and initialized with default `Handler` objects specified in the logging configuration file as described above. This root logger has no name, and you can obtain a reference to it by passing the empty string to the `getLogger()` method. Except for this root logger and anonymous loggers (see `Logger.getAnonymousLogger()`), all loggers have names, and they are typically named after the package or class for which they provide logging. When a named logger is registered with the `LogManager`, the `LogManager` examines its name and inserts it into the tree of loggers at the appropriate place: a logger named “java.util.logging” would be inserted as the child of a logger named “java.util”, if any such logger existed, or as a child of a logger named “java”, or, if no logger with that name existed either, it would be inserted as a child of the root logger named “”. When the `LogManager` determines the position of a logger within the tree of loggers, it calls the `setParent()` method of the newly-registered `Logger` to tell it who its parent is. This is important because, by default, loggers inherit their logging level and handlers from their parent. Although the `Logger.setParent()` method is public, it is intended for use only by the `LogManager` class.

Anonymous loggers created with `Logger.getAnonymousLogger()` do not have names, and are not part of the logger tree. When they are created, however, their parent is set to the root logger of the `LogManager`. For this reason, anonymous loggers inherit the default handlers specified in the logging configuration file.

The `readConfiguration()` methods are used to force the `LogManager` to re-read the system configuration file, or to read a new configuration file from the specified stream. Both versions of the method generate a `java.beans.PropertyChangeEvent` and use it to notify any listeners that have been registered with `addPropertyChangeListener`. Both methods also first invoke the `reset()` method which discards the properties of the current configuration file, removes and closes all handlers for all loggers, and sets the logging level of all loggers to null, except for the root logger's logging level, which it sets to `Level.INFO`. It is unlikely

## LogManager

that you would ever want to invoke `reset()` yourself. A number of `LogManager` methods throw a `SecurityException` if the caller does not have appropriate permissions. You can use `checkAccess()` to test whether the current calling context has the required `LoggingPermission` named “control”.

All `LogManager` methods can be safely used by multiple threads.

```
public class LogManager {
    // Protected Constructors
    protected LogManager();
    // Public Class Methods
    public static LogManager getLogManager();
    // Event Registration Methods (by event name)
    public void addPropertyChangeListener(java.beans.PropertyChangeListener l) throws SecurityException;
    public void removePropertyChangeListener(java.beans.PropertyChangeListener l) throws SecurityException;
    // Public Instance Methods
    public boolean addLogger(Logger logger); synchronized
    public void checkAccess() throws SecurityException;
    public Logger getLogger(String name); synchronized
    public java.util.Enumeration getLoggerNames(); synchronized
    public String getProperty(String name);
    public void readConfiguration() throws java.io.IOException, SecurityException;
    public void readConfiguration(java.io.InputStream ins) throws java.io.IOException, SecurityException;
    public void reset() throws SecurityException;
}
```

Returned By: `LogManager.getLogManager()`

## LogRecord

Java 1.4

`java.util.logging`

*serializable*

Instances of this class are used to represent log messages as they are passed between `Logger`, `Handler`, `Filter` and `Formatter` objects. `LogRecord` defines a number of JavaBeans-type property getter and setter methods. The values of the various properties encapsulate all details of the log message. The `LogRecord()` constructor takes arguments for the two most important properties: the log level and the log message (or localization key). The constructor also initializes the `millis` property to the current time, the `sequenceNumber` property to a unique (within the VM) value that can be used to compare the order of two log messages, and the `threadID` property to a unique identifier for the current thread. All other properties of the `LogRecord` are left uninitialized with their default null values.

Object	LogRecord	Serializable
--------	-----------	--------------

```
public class LogRecord implements Serializable {
    // Public Constructors
    public LogRecord(Level level, String msg);
    // Property Accessor Methods (by property name)
    public Level getLevel();
    public void setLevel(Level level);
    public String getLoggerName();
    public void setLoggerName(String name);
    public String getMessage();
    public void setMessage(String message);
    public long getMillis();
    public void setMillis(long millis);
    public Object[] getParameters();
    public void setParameters(Object[] parameters);
}
```



```

public java.util.ResourceBundle getResourceBundle();
public void setResourceBundle(java.util.ResourceBundle bundle);
public String getResourceBundleName();
public void setResourceBundleName(String name);
public long getSequenceNumber();
public void setSequenceNumber(long seq);
public String getSourceClassName();
public void setSourceClassName(String sourceClassName);
public String getSourceMethodName();
public void setSourceMethodName(String sourceMethodName);
public int getThreadID();
public void setThreadID(int threadID);
public Throwable getThrown();
public void setThrown(Throwable thrown);
}

```

*Passed To:* ConsoleHandler.publish(), FileHandler.publish(), Filter.isLoggable(), Formatter.format(), formatMessage(), Handler.isLoggable(), publish(), Logger.log(), MemoryHandler.isLoggable(), publish(), SimpleFormatter.format(), SocketHandler.publish(), StreamHandler.isLoggable(), publish(), XMLFormatter.format()

## MemoryHandler

Java 1.4

java.util.logging

A `MemoryHandler` stores `LogRecord` objects in a fixed-sized buffer in memory. When the buffer fills up, it discards the oldest record one each time a new record arrives. It maintains a reference to another `Handler` object, and whenever the `push()` method is called, or whenever a `LogRecord` arrives with a level at or higher than the `pushLevel` threshold, it “pushes” all of buffered `LogRecord` objects to that other `Handler` object, which typically formats and outputs them to some appropriate destination. Because `MemoryHandler` never outputs log records itself, it does not use the `formatter` or `encoding` properties inherited from its superclass.

When you create a `MemoryHandler`, you can specify the target `Handler` object, the size of the in-memory buffer, and the value of the `pushLevel` property, or you can omit these constructor arguments and rely on system-wide defaults obtained with `LogManager.getProperty()`. `MemoryHandler` also uses `LogManager.getProperty()` to obtain initial values for the level and filter properties inherited from `Handler`. The following table lists these properties, as well as the `target`, `size`, and `pushLevel` constructor arguments, the value passed to `getProperty()`, and the default value used if `getProperty()` returns null. See `Handler` for further details.

Property or argument	LogManager property name	Default
level	java.util.logging.MemoryHandler.level	Level.ALL
filter	java.util.logging.MemoryHandler.filter	null
target	java.util.logging.MemoryHandler.target	No default
size	java.util.logging.MemoryHandler.size	1,000 log records
pushLevel	java.util.logging.MemoryHandler.push	Level.SEVERE



```

public class MemoryHandler extends Handler {
// Public Constructors

```

## MemoryHandler

```
public MemoryHandler();
public MemoryHandler(Handler target, int size, Level pushLevel);
// Public Instance Methods
public Level getPushLevel();                                synchronized
public void push();                                          synchronized
public void setPushLevel(Level newLevel) throws SecurityException;
// Public Methods Overriding Handler
public void close() throws SecurityException;
public void flush();
public boolean isLoggable(LogRecord record);
public void publish(LogRecord record);                        synchronized
}
```

## SimpleFormatter

Java 1.4

java.util.logging

This **Formatter** subclass converts a **LogRecord** object to a human-readable log message that is typically one or two lines long. See also **XMLFormatter**.

Object — Formatter — SimpleFormatter

```
public class SimpleFormatter extends Formatter {
// Public Constructors
public SimpleFormatter();
// Public Methods Overriding Formatter
public String format(LogRecord record);                        synchronized
}
```

## SocketHandler

Java 1.4

java.util.logging

This **Handler** subclass formats **LogRecord** objects and outputs the resulting strings to a network socket. When you create a **SocketHandler**, you can pass the hostname and port of the socket to the constructor or you can rely on system-wide defaults obtained with **LogManager.getProperty()**. **SocketHandler** also uses **LogManager.getProperty()** to obtain initial values for the properties inherited from **Handler**. The following table lists these properties, as well as the host and port arguments, the value passed to **getProperty()**, and the default value used if **getProperty()** returns null. See **Handler** for further details.

Handler property	LogManager property name	Default
level	java.util.logging.SocketHandler.level	Level.ALL
filter	java.util.logging.SocketHandler.filter	null
formatter	java.util.logging.SocketHandler.formatter	XMLFormatter
encoding	java.util.logging.SocketHandler.encoding	Platform default
hostname	java.util.logging.SocketHandler.host	No default
port	java.util.logging.SocketHandler.port	No default

Object — Handler — StreamHandler — SocketHandler

```
public class SocketHandler extends StreamHandler {
// Public Constructors
public SocketHandler() throws java.io.IOException;
```

```

    public SocketHandler(String host, int port) throws java.io.IOException;
    // Public Methods Overriding StreamHandler
    public void close() throws SecurityException; synchronized
    public void publish(LogRecord record); synchronized
}

```

## StreamHandler

Java 1.4

java.util.logging

This Handler subclass sends log messages to an arbitrary java.io.OutputStream. It exists primarily to serve as the common superclass of ConsoleHandler, FileHandler, and SocketHandler.

Object	Handler	StreamHandler
--------	---------	---------------

```

public class StreamHandler extends Handler {
    // Public Constructors
    public StreamHandler();
    public StreamHandler(java.io.OutputStream out, Formatter formatter);
    // Public Methods Overriding Handler
    public void close() throws SecurityException; synchronized
    public void flush(); synchronized
    public boolean isLoggable(LogRecord record);
    public void publish(LogRecord record); synchronized
    public void setEncoding(String encoding) throws SecurityException, java.io.UnsupportedEncodingException;
    // Protected Instance Methods
    protected void setOutputStream(java.io.OutputStream out) throws SecurityException; synchronized
}

```

*Subclasses:* ConsoleHandler, FileHandler, SocketHandler

## XMLFormatter

Java 1.4

java.util.logging

This Formatter subclass converts a LogRecord to an XML-formatted string. The format() method returns a <record> element, which always contains <date>, <millis>, <sequence>, <level> and <message> tags, and may also contain <logger>, <class>, <method>, <thread>, <key>, <catalog>, <param>, and <exception> tags. See <http://java.sun.com/dtd/logger.dtd> for the DTD of the output document.

The getHead() and getTail() methods are overridden to return opening and closing <log> and </log> tags to surround all output <record> tags. Note however, that if an application terminates abnormally, the logging facility may be unable to terminate the log file with the closing <log> tag.

Object	Formatter	XMLFormatter
--------	-----------	--------------

```

public class XMLFormatter extends Formatter {
    // Public Constructors
    public XMLFormatter();
    // Public Methods Overriding Formatter
    public String format(LogRecord record);
    public String getHead(Handler h);
    public String getTail(Handler h);
}

```

## Package **java.util.prefs**

**Java 1.4**

The `java.util.prefs` package contains classes and interfaces for managing persistent user and system-wide preferences for Java applications and classes. Most applications will use only the `Preferences` class itself. Some will also use the event objects and listener interfaces defined by this package, and some may need to explicitly catch the types of exceptions defined by this package. Application programmers never need to use the `PreferencesFactory` interface or the `AbstractPreferences` class, which are intended for Preferences implementors only.

To use the `Preferences` class, first use a static method to obtain an appropriate `Preferences` object or objects, and then use a `get()` method to query a preference value or a `put()` method to set a preference value. The following code shows a typical usage. See the `Preferences` class for details.

```
import java.util.prefs.Preferences;

public class TextEditor {
    // Some constants that define default values for preferences
    public static final int WIDTH_DEFAULT = 80;
    public static final String DICTIONARY_DEFAULT = "";

    // Fields to be initialized from preference values
    public int width;           // Screen width in columns
    public String dictionary;   // Dictionary name for spell checking

    public void initPrefs() {
        // Get Preferences objects for user and system preferences for this package
        Preferences userprefs = Preferences.userNodeForPackage(TextEditor.class);
        Preferences sysprefs = Preferences.systemNodeForPackage(TextEditor.class);

        // Look up preference values. Note that you always pass a default value.
        width = userprefs.getInt("width", WIDTH_DEFAULT);
        // Look up a user preference using a system preference as the default
        dictionary = userprefs.get("dictionary",
                                   sysprefs.get("dictionary",
                                                DICTIONARY_DEFAULT));
    }
}
```

### *Interface:*

```
public interface PreferencesFactory;
```

### *Events:*

```
public class NodeChangeEvent extends java.util.EventObject;
```

```
public class PreferenceChangeEvent extends java.util.EventObject;
```

### *Event Listeners:*

```
public interface NodeChangeListener extends java.util.EventListener;
```

```
public interface PreferenceChangeListener extends java.util.EventListener;
```

*Other Classes:*

```
public abstract class Preferences;
public abstract class AbstractPreferences extends Preferences;
```

*Exceptions:*

```
public class BackingStoreException extends Exception;
public class InvalidPreferencesFormatException extends Exception;
```

**AbstractPreferences**

Java 1.4

java.util.prefs

This class implements all the abstract methods of `Preferences` on top of a smaller set of abstract methods. Programmers creating a `Preferences` implementation (or “service provider”) can subclass this class and need define only the nine methods whose names end in “Spi”. Application programmers never need to use this class.

Object	Preferences	AbstractPreferences
--------	-------------	---------------------

```
public abstract class AbstractPreferences extends Preferences {
// Protected Constructors
    protected AbstractPreferences(AbstractPreferences parent, String name);
// Event Registration Methods (by event name)
    public void addNodeChangeListener(NodeChangeListener ncl);           Overrides:Preferences
    public void removeNodeChangeListener(NodeChangeListener ncl);       Overrides:Preferences
    public void addPreferenceChangeListener(PreferenceChangeListener pcl); Overrides:Preferences
    public void removePreferenceChangeListener(PreferenceChangeListener pcl); Overrides:Preferences
// Public Methods Overriding Preferences
    public String absolutePath();
    public String[] childrenNames() throws BackingStoreException;
    public void clear() throws BackingStoreException;
    public void exportNode(java.io.OutputStream os) throws java.io.IOException, BackingStoreException;
    public void exportSubtree(java.io.OutputStream os) throws java.io.IOException, BackingStoreException;
    public void flush() throws BackingStoreException;
    public String get(String key, String def);
    public boolean getBoolean(String key, boolean def);
    public byte[] getByteArray(String key, byte[] def);
    public double getDouble(String key, double def);
    public float getFloat(String key, float def);
    public int getInt(String key, int def);
    public long getLong(String key, long def);
    public boolean isUserNode();
    public String[] keys() throws BackingStoreException;
    public String name();
    public Preferences node(String path);
    public boolean nodeExists(String path) throws BackingStoreException;
    public Preferences parent();
    public void put(String key, String value);
    public void putBoolean(String key, boolean value);
    public void putByteArray(String key, byte[] value);
```

## AbstractPreferences

```
public void putDouble(String key, double value);
public void putFloat(String key, float value);
public void putInt(String key, int value);
public void putLong(String key, long value);
public void remove(String key);
public void removeNode() throws BackingStoreException;
public void sync() throws BackingStoreException;
public String toString();
// Protected Instance Methods
protected final AbstractPreferences[] cachedChildren();
protected abstract String[] childrenNamesSpi() throws BackingStoreException;
protected abstract AbstractPreferences childSpi(String name);
protected abstract void flushSpi() throws BackingStoreException;
protected abstract AbstractPreferences getChild(String nodeName) throws BackingStoreException;
protected abstract String getSpi(String key);
protected boolean isRemoved();
protected abstract String[] keysSpi() throws BackingStoreException;
protected abstract void putSpi(String key, String value);
protected abstract void removeNodeSpi() throws BackingStoreException;
protected abstract void removeSpi(String key);
protected abstract void syncSpi() throws BackingStoreException;
// Protected Instance Fields
protected final Object lock;
protected boolean newNode;
}
```

*Passed To:* AbstractPreferences.AbstractPreferences()

*Returned By:* AbstractPreferences.{cachedChildren(), childSpi(), getChild()}

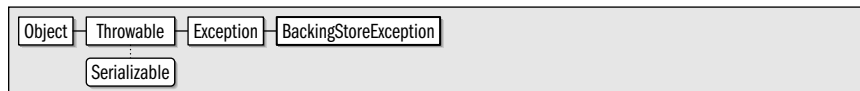
## BackingStoreException

Java 1.4

java.util.prefs

serializable checked

This exception signals that a Preferences method could not complete because of an implementation-specific problem with the preferences database. The most commonly used methods of the Preferences class do not throw this exception, and are guaranteed to succeed even if the implementation's preferences data is not available. Note that although this class inherits the Serializable interface, implementations are not actually required to be serializable.



```
public class BackingStoreException extends Exception {
// Public Constructors
public BackingStoreException(Throwable cause);
public BackingStoreException(String s);
}
```

*Thrown By:* Too many methods to list.

**InvalidPreferencesFormatException**

Java 1.4

java.util.prefs

serializable checked

This exception signals a syntax error in XML preference data. Note that although this class inherits the `Serializable` interface, implementations are not actually required to be serializable.



```

public class InvalidPreferencesFormatException extends Exception {
// Public Constructors
    public InvalidPreferencesFormatException(String message);
    public InvalidPreferencesFormatException(Throwable cause);
    public InvalidPreferencesFormatException(String message, Throwable cause);
}
  
```

*Thrown By:* Preferences.importPreferences()

**NodeChangeEvent**

Java 1.4

java.util.prefs

serializable event

A `NodeChangeEvent` object is passed to the methods of any `NodeChangeListener` objects registered on a `Preferences` object when a child `Preferences` node is added or removed. `getChild()` returns the `Preferences` object that was added or removed. `getParent()` returns the parent `Preferences` node from which the child was added or removed. This parent `Preferences` object is the one on which the `NodeChangeListener` was registered.

Although this class inherits the `Serializable` interface, it is not actually serializable.



```

public class NodeChangeEvent extends java.util.EventObject {
// Public Constructors
    public NodeChangeEvent(Preferences parent, Preferences child);
// Public Instance Methods
    public Preferences getChild();
    public Preferences getParent();
}
  
```

*Passed To:* NodeChangeListener.{childAdded(), childRemoved()}

**NodeChangeListener**

Java 1.4

java.util.prefs

event listener

This interface defines the methods that an object must implement if it wants to be notified when a child preferences node is added to or removed from a `Preferences` object. When such an addition or removal occurs, the parent `Preferences` object passes a `NodeChangeEvent` object to the appropriate method of any `NodeChangeListener` objects that have been registered through the `Preferences.addNodeChangeListener()` method.



```

public interface NodeChangeListener extends java.util.EventListener {
// Public Instance Methods
}
  
```

## NodeChangeListener

```
public abstract void childAdded(NodeChangeEvent evt);
public abstract void childRemoved(NodeChangeEvent evt);
}
```

*Passed To:* AbstractPreferences.{addNodeChangeListener(), removeNodeChangeListener()}, Preferences.{addNodeChangeListener(), removeNodeChangeListener()}

## PreferenceChangeEvent

Java 1.4

java.util.prefs

serializable event

A PreferenceChangeEvent object is passed to the preferenceChange() method of any PreferenceChangeListener objects registered on a Preferences object whenever a preferences value is added to, removed from, or modified in that Preferences node. getNode() returns the affected Preferences object. getKey() returns name of the modified preference. If the preference value was added or modified, getNewValue() returns that value. If a preference was deleted, getNewValue() returns null.

Although this class inherits the Serializable interface, it is not actually serializable.



```
public class PreferenceChangeEvent extends java.util.EventObject {
    // Public Constructors
    public PreferenceChangeEvent(Preferences node, String key, String newValue);
    // Public Instance Methods
    public String getKey();
    public String getNewValue();
    public Preferences getNode();
}
```

*Passed To:* PreferenceChangeListener.preferenceChange()

## PreferenceChangeListener

Java 1.4

java.util.prefs

event listener

This interface defines the method that an object must implement if it wants to be notified when a preference key/value pair is added to, removed from, or changed in a Preferences object. After any such change, the Preferences object passes a PreferenceChangeEvent object describing the change to the preferenceChange() method of any PreferenceChangeListener objects that have been registered through the Preferences.addPreferenceChangeListener() method.



```
public interface PreferenceChangeListener extends java.util.EventListener {
    // Public Instance Methods
    public abstract void preferenceChange(PreferenceChangeEvent evt);
}
```

*Passed To:* AbstractPreferences.{addPreferenceChangeListener(), removePreferenceChangeListener()}, Preferences.{addPreferenceChangeListener(), removePreferenceChangeListener()}



**Preferences****Java 1.4****java.util.prefs**

A **Preferences** object represents a mapping between preference names, which are case-sensitive strings, and corresponding preference values. **get()** allows you to query the string value of a named preference, and **put()** allows you to set a string value for a named preference. Although all preference values are stored as strings, various convenience methods whose names begin with “get” and “put” exist to convert preference values of type **boolean**, **byte[]**, **double**, **float**, **int**, and **long** to and from strings.

The **remove()** method allows you to delete a named preference altogether, and **clear()** deletes all preference values stored in a **Preferences** object. The **keys()** method returns an array of strings that specify the names of all preferences in the **Preferences** object.

Preference values are stored in some implementation-dependent back-end which may be a file, a LDAP directory server, the Windows Registry, or any other persistent “backing store.” Note that all the **get()** methods of this class require a default value to be specified. They return this default if no value has been stored for the named preference, or if the backing store is unavailable for any reason. The **Preferences** class is completely independent of the underlying implementation, except that it enforces an 80-character limit for preference names and **Preference** node names (see the following) and a 8,192-character limit on preference value strings.

**Preferences** does not have a public constructor. To obtain a **Preferences** object for use in your application, you must use one of the static methods described below. Each **Preferences** object is a node in a hierarchy of **Preferences** nodes. There are two distinct hierarchies: one stores user-specific preferences, and one stores system-wide preferences. All **Preferences** nodes (in either hierarchy) have a unique name and use the same naming convention that Unix filesystems use. Applications (and classes) may store their preferences in a **Preferences** node with any name, but the convention is to use a node name that corresponds to the package name of the application or class, with all “.” characters in the package name converted to “/” characters. For example, the preferences node used by **java.lang.System** would be “/java/lang”.

**Preferences** defines static methods that you can use to obtain the **Preferences** objects your application requires. Pass a **Class** object to **systemNodeForPackage()** and **userNodeForPackage()** to obtain the system and user **Preferences** objects that are specific to the package of that class. If you want a **Preferences** node specific to a single class rather than to the package, you can pass the class name to the **node()** method of the package-specific node returned by **systemNodeForPackage()** or **userNodeForPackage()**. If you want to navigate the entire tree of preferences nodes (which most applications never need to do) call **systemRoot()** and **userRoot()** to obtain the root node of the two hierarchies, and then use the **node()** method to look up child nodes of those roots.

Various **Preferences** methods allow you to traverse the preferences hierarchies. **parent()** returns the parent **Preferences** node. **childrenNames()** returns an array of the relative names of all children of a **Preferences** node. **node()** returns a named **Preferences** object from the hierarchy. If the specified node name begins with a slash, it is an absolute name and is interpreted relative to the root of the hierarchy. Otherwise, it is a relative name and is interpreted relative to the **Preferences** object on which **node()** was called. **nodeExists()** allows you to test whether a named node exists. **removeNode()** allows you to delete an entire **Preferences** node from the hierarchy (useful when uninstalling an application). **name()** returns the simple name of a **Preferences** node, relative to its parent. **absolutePath()** returns the full, absolute name of the node, relative to the root of the hierarchy. Finally, **isUserNode()** allows you to determine whether a **Preferences** object is part of the user or system hierarchies.

## Preferences

Many applications will simply read their preference values once at startup. Long-lived applications or applications that want to respond dynamically to modifications to preferences (such as applications that are tightly integrated with a graphical desktop) may use `addPreferenceChangeListener()` to register a `PreferenceChangeListener` to receive notifications of preference changes (in the form of `PreferenceChangeEvent` objects). Applications that are interested in changes to the `Preferences` hierarchy itself can register a `NodeChangeListener`.

`put()` and the various type-specific `put...()` convenience methods may return asynchronously, before the new preference value is stored persistently within the backing store. Call `flush()` to force any preference changes to this `Preferences` node (and any of its descendants in the hierarchy) to be stored persistently. Note that it is not necessary to call `flush()` before an application terminates; all preferences will eventually be made persistent. More than one application (within more than one Java virtual machine) may set preference values in the same `Preferences` node at the same time. Call `sync()` to ensure that future calls to `get()` and its related convenience methods retrieve current preference values set by this or other virtual machines. Note that the `flush()` and `sync()` operations are typically much more expensive than `get()` and `put()` operations, and applications do not often need to use them.

`Preferences` implementations ensure that all the methods of this class are thread safe. If multiple threads or multiple VMs write store the same preferences concurrently, their values may overwrite one another, but the preference data will not be corrupted. Note that, for simplicity, `Preferences` does not define any way to set multiple preferences in a single atomic transaction. If you need to ensure atomicity for multiple preference values, define a data format that allows you to store all the requisite values in a single string, and set and query those values with a single call to `put()` or `get()`.

The contents of a `Preferences` node, or of a node and all of its descendants may be exported as an XML file with `exportNode()` and `exportSubtree()`. The static `importPreferences()` method reads an exported XML file back into the preferences hierarchy. These methods allow backups to be made of preference data, and allow preferences to be transferred between systems or between users.

Prior to Java 1.4, application preferences were sometimes managed with the `java.util.Properties` object.

```
public abstract class Preferences {
    // Protected Constructors
    protected Preferences();
    // Public Constants
    public static final int MAX_KEY_LENGTH;           =80
    public static final int MAX_NAME_LENGTH;         =80
    public static final int MAX_VALUE_LENGTH;        =8192
    // Public Class Methods
    public static void importPreferences(java.io.InputStream is) throws java.io.IOException,
        InvalidPreferencesFormatException;
    public static Preferences systemNodeForPackage(Class c);
    public static Preferences systemRoot();
    public static Preferences userNodeForPackage(Class c);
    public static Preferences userRoot();
    // Event Registration Methods (by event name)
    public abstract void addNodeChangeListener(NodeChangeListener ncl);
    public abstract void removeNodeChangeListener(NodeChangeListener ncl);
    public abstract void addPreferenceChangeListener(PreferenceChangeListener pcl);
    public abstract void removePreferenceChangeListener(PreferenceChangeListener pcl);
    // Public Instance Methods
```

```

public abstract String absolutePath();
public abstract String[] childrenNames() throws BackingStoreException;
public abstract void clear() throws BackingStoreException;
public abstract void exportNode(java.io.OutputStream os) throws java.io.IOException, BackingStoreException;
public abstract void exportSubtree(java.io.OutputStream os) throws java.io.IOException, BackingStoreException;
public abstract void flush() throws BackingStoreException;
public abstract String get(String key, String def);
public abstract boolean getBoolean(String key, boolean def);
public abstract byte[] getByteArray(String key, byte[] def);
public abstract double getDouble(String key, double def);
public abstract float getFloat(String key, float def);
public abstract int getInt(String key, int def);
public abstract long getLong(String key, long def);
public abstract boolean isUserNode();
public abstract String[] keys() throws BackingStoreException;
public abstract String name();
public abstract Preferences node(String pathName);
public abstract boolean nodeExists(String pathName) throws BackingStoreException;
public abstract Preferences parent();
public abstract void put(String key, String value);
public abstract void putBoolean(String key, boolean value);
public abstract void putByteArray(String key, byte[] value);
public abstract void putDouble(String key, double value);
public abstract void putFloat(String key, float value);
public abstract void putInt(String key, int value);
public abstract void putLong(String key, long value);
public abstract void remove(String key);
public abstract void removeNode() throws BackingStoreException;
public abstract void sync() throws BackingStoreException;
// Public Methods Overriding Object
public abstract String toString();
}

```

*Subclasses:* AbstractPreferences

*Passed To:* NodeChangeEvent.NodeChangeEvent(), PreferenceChangeEvent.PreferenceChangeEvent()

*Returned By:* AbstractPreferences.{node(), parent()}, NodeChangeEvent.{getChild(), getParent()}, PreferenceChangeEvent.getNode(), Preferences.{node(), parent(), systemNodeForPackage(), systemRoot(), userNodeForPackage(), userRoot()}, PreferencesFactory.{systemRoot(), userRoot()}

## PreferencesFactory

Java 1.4

java.util.prefs

The PreferencesFactory interface defines the factory methods used by the static methods of the Preferences class to obtain the root Preferences nodes for user-specific and system-wide preferences hierarchies. Application programmers never need to use this interface.

An implementation of the preferences API for a specific back-end data store must include an implementation of this interface that works with that data store. Sun's implementation of Java includes a default filesystem-based implementation, which you can override by specifying the name of a PreferencesFactory implementation as the value of the java.util.prefs.PreferencesFactory system property.

```

public interface PreferencesFactory {
// Public Instance Methods
public abstract Preferences systemRoot();
}

```

## *PreferencesFactory*

```
public abstract Preferences userRoot();  
}
```

## **Package java.util.regex**

**Java 1.4**

This small package provides a facility for textual pattern matching with regular expressions. **Pattern** objects represent regular expressions, which are specified using a syntax very close to the one used by the Perl programming language. The **Matcher** class encapsulates a **Pattern** and a string of text, and defines various methods for matching the pattern to the text. The methods of the **Pattern** and **Matcher** classes that specify text to be matched all specify that text in the form of a `java.lang.CharSequence`. The `CharSequence` interface is new in Java 1.4. It is implemented by the `String` and `StringBuffer` classes, and also by the `java.nio.CharBuffer` class of the New I/O API.

In addition to the pattern matching methods defined in this package, the `java.lang.String` class has been augmented in Java 1.4 with a number of convenience methods for matching strings against regular expressions that are specified in their text form as strings, rather than in their compiled form as **Pattern** objects. Applications with simple pattern matching needs can use these convenience methods and may never have to directly use the **Pattern** or **Matcher** classes.

### *Classes:*

```
public final class Matcher;  
public final class Pattern implements Serializable;
```

### *Exceptions:*

```
public class PatternSyntaxException extends IllegalArgumentException;
```

## **Matcher**

**Java 1.4**

**java.util.regex**

A **Matcher** object encapsulates a regular expression and a string of text (a **Pattern** and a `java.lang.CharSequence`) and defines methods for matching the pattern to the text in several different ways, for obtaining details about pattern matches, and for doing search-and-replace operations on the text. **Matcher** has no public constructor. Obtain a **Matcher** by passing the character sequence to be matched to the `matcher()` method of the desired **Pattern** object. You can also reuse an existing **Matcher** object with a new character sequence (but the same **Pattern**) by passing a new `CharSequence` to the `matcher`'s `reset()` method.

Once you have created or reset a **Matcher**, there are several types of comparisons you can perform between the regular expression and the character sequence. The simplest comparison is the `matches()` method. It returns `true` if the pattern matches the complete character sequence, and returns `false` otherwise. The `lookingAt()` method is similar: it returns `true` if the pattern matches the complete sequence, or if it matches some subsequence at the beginning of the text. If the pattern does not match the start of the text, `lookingAt()` returns `false`. `matches()` requires the pattern to match both the beginning and ending of the text, and `lookingAt()` requires the pattern to match the beginning. The `find()` method, on the other hand, has neither of these requirements: it returns `true` if the pattern matches any part of the text. As will be described below, `find()` has some special behavior that allows it to be used in a loop to find all matches in the text.

If `matches()`, `lookingAt()`, or `find()` return `true`, then several other **Matcher** methods can be used to obtain details about the matched text. The no-argument `start()` method returns

the index of the first character that matched the pattern (for `matches()` and `lookingAt()` this must be zero, of course.) The no-argument `end()` method returns the index of the last character that matched plus one (or the index of the first character following the matched text). Some regular expressions can match the empty string. If this occurs, then `end()` returns the same value as `start()`.

Regular expressions may include subexpressions grouped within parentheses, which are also known as “capturing groups”. When working with regular expressions, it is often useful to obtain information about the text that matches these groups. The one-argument versions of the `start()` and `end()` methods take a group number and return the index of the first character that matched that group and the index of the first character following the text that matched that group. Similarly, the `group()` method takes a group number and returns a `String` that contains the text that matched that group. (The no-argument version of `group()` returns the text that matched the entire regular expression.) Groups are numbered from 1 (group 0 is the entire regular expression) and are ordered from left-to-right within the regular expression. When there are nested groups, their ordering is based on the position of the opening left parenthesis that begins the group.

The no-argument version of `find()` has special behavior that makes it suitable for use in a loop to find all matches of a pattern within a string. The first time `find()` is called after a `Matcher` is created or after the `reset()` method is called, it starts its search at the beginning of the string. If it finds a match, it stores the start and end position of the matched text. If `reset()` is not called in the meantime, then the next call to `find()` searches again but starts the search at the first character after the match: at the position returned by `end()`. (If the previous call to `find()` matched the empty string, then the next call begins at `end()+1` instead.) In this way, it is possible to find all matches of a pattern within a string simply by calling `find()` repeatedly until it returns `false` indicating that no match was found. After each repeated call to `find()` you can use the `start()`, `end()` and `group()` methods to obtain more information about the text that matched the pattern and any of its subpatterns.

`Matcher` also defines methods that perform search-and-replace operations. `replaceFirst()` searches the character sequence for the first subsequence that matches the pattern. It then returns a string that is the character sequence with the matched text replaced with the specified replacement string. `replaceAll()` is similar, but replaces all matching subsequences within the character sequence instead of just replacing the first. The replacement string passed to `replaceFirst()` and `replaceAll()` is not always replaced literally. If the replacement contains a dollar sign followed by an integer that is a valid group number, then the dollar sign and the number are replaced by the text that matched the numbered group. If you want to include a literal dollar sign in the replacement string, precede it with a backslash.

`replaceFirst()` and `replaceAll()` are convenience methods that cover the most common search-and-replace cases. However, `Matcher` also defines lower-level methods that you can use to do a custom search-and-replace operation in conjunction with calls to `find()`, and build up a modified string in a `StringBuffer`. In order to understand this search-and-replace procedure, you must know that a `Matcher` maintains a “append position”, which starts at zero when the `Matcher` is created, and is restored to zero by the `reset()` method. The `appendReplacement()` method is designed to be used after a successful call to `find()`. It copies all the text between the append position and the character before the `start()` position for the last match into the specified string buffer. Then it appends the specified replacement text to that string buffer (performing the same substitutions that `replaceAll()` does). Finally, it sets the append position to the `end()` of the last match, so that a subsequent call to `appendReplacement()` starts at a new character. `appendReplacement()` is intended for use after a call to `find()` that returns `true`. When `find()` cannot find another match and returns `false`, you should complete the replacement operation by calling

## Matcher

`appendTail()`: this method copies all text between the `end()` position of the last match and the end of the character sequence into the specified `StringBuffer`.

The `reset()` method has been mentioned several times. It erases any saved information about the last match, and restores the `Matcher` to its initial state so that subsequent calls to `find()` and `appendReplacement()` start at the beginning of the character sequence. The one-argument version of `reset()` also allows you to specify an entirely new character sequence to match against. It is important to understand that several other `Matcher` methods call `reset()` themselves before they perform their operation. They are: `matches()`, `lookingAt()`, the one-argument version of `find()`, `replaceAll()`, and `replaceFirst()`.

`Matcher` is not thread-safe, and should not be used by more than one thread concurrently.

```
public final class Matcher {  
    // No Constructor  
    // Public Instance Methods  
    public Matcher appendReplacement(StringBuffer sb, String replacement);  
    public StringBuffer appendTail(StringBuffer sb);  
    public int end();  
    public int end(int group);  
    public boolean find();  
    public boolean find(int start);  
    public String group();  
    public String group(int group);  
    public int groupCount();  
    public boolean lookingAt();  
    public boolean matches();  
    public Pattern pattern();  
    public String replaceAll(String replacement);  
    public String replaceFirst(String replacement);  
    public Matcher reset();  
    public Matcher reset(CharSequence input);  
    public int start();  
    public int start(int group);  
}
```

*Returned By:* `Matcher.{appendReplacement(), reset()}, Pattern.matcher()`

## Pattern

Java 1.4

`java.util.regex`

*serializable*

This class represents a regular expression. It has no public constructor; obtain a `Pattern` by calling one of the static `compile()` methods, passing the string representation of the regular expression, and an optional bitmask of flags that modify the behavior of the regex. `pattern()` and `flags()` return the string form of the regular expression and the bitmask that were passed to `compile()`.

If you want to perform only a single match operation with a regular expression, and don't need to use any of the flags, you don't have to create a `Pattern` object: simply pass the string representation of the pattern and the `CharSequence` to be matched to the static `matches()` method: the method returns `true` if the specified pattern matches the complete specified text, or returns `false` otherwise.

`Pattern` represents a regular expression, but does not actually define any primitive methods for matching regular expressions to text. To do that, you must create a `Matcher` object that encapsulates a pattern and the text it is to be compared with. Do this by calling the `matcher()` method and specifying the `CharSequence` you want to match against. See `Matcher` for a description of what you can do with it.

The `split()` methods are the exception to the rule that you must obtain a `Matcher` in order to be able to do anything with a `Pattern` (although they create and use a `Matcher` internally). They take a `CharSequence` as input, and split it into substrings, using text that matches the regular expression as the delimiter, returning the substrings as a `String[]`. The two-argument version of `split()` takes an integer argument that specifies the maximum number of substrings to break the input into.

`Pattern` defines the following flags that control various aspects of how regular expression matching is performed. The flags are the following:

#### **CANON\_EQ**

The Unicode standard sometimes allows more than one way to specify the same character. If this flag is set, characters are compared by comparing their full canonical decompositions, so that characters will match even if expressed in different ways. Enabling this flag typically slows down performance. Unlike all the other flags, there is no way to temporarily enable this flag within a pattern.

#### **CASE\_INSENSITIVE**

Match letters without regard to case. By default this flag only affects the comparisons of ASCII letters. Also set the `UNICODE_CASE` flag if you want to ignore the case of all Unicode characters. You can enable this flag within a pattern with `(?i)`.

#### **COMMENTS**

If this flag is set, then whitespace and comments within a pattern are ignored. Comments are all characters between a `#` and end of line. You can enable this flag within a pattern with `(?x)`.

#### **DOTALL**

If this flag is set, then the `.` expression matches any character. If it is not set, then it does not match line terminator characters. This is also known as single-line mode, and you can enable it within a pattern with `(?s)`.

#### **MULTILINE**

If this flag is set, then the `^` and `$` anchors match not only at the beginning and end of the input string, but also at the beginning and end of any lines within that string. Within a pattern you can enable this flag with `(?m)`.

#### **UNICODE\_CASE**

If this flag is set along with the `CASE_INSENSITIVE` flag, then case-insensitive comparison is done for all Unicode letters, rather than just for ASCII letters. You can enable both flags within a pattern with `(?iu)`.

#### **UNIX\_LINES**

If this flag is set, then only the newline character is considered a line terminator for the purposes of `.`, `^`, and `$`. If the flag is not set, then newlines (`\n`), carriage returns (`\r`), and carriage return newline sequences (`\r\n`) are all considered line terminators, as are the Unicode characters, `\u0085` (“next line”), `\u2028` (“line separator”), and `\u2029` (“paragraph separator”). You can turn this flag on within a pattern with `(?d)`.

Although the API for the `Pattern` class is quite simple, the syntax for the text representation of regular expressions is fairly complex. A complete tutorial on regular expressions is beyond the scope of this book. Table 27-1 is a quick reference for regular expression

## Pattern

syntax. It is very similar to the syntax used in Perl. Note that many of the syntax elements of a regular expression include a backslash character, such as `\d` to match one of the digits 0–9. Because Java strings also use the backslash character as an escape, you must double the backslashes when expressing a regular expression as a string literal: `"\\d"`. For complete details on regular expressions, see *Programming Perl* or *Mastering Regular Expressions* (both by O'Reilly).

Table 17–1. Java regular expression quick reference

Syntax	Matches
<b>Single characters</b>	
<code>x</code>	The character <code>x</code> , as long as <code>x</code> is not a punctuation character with special meaning in the regular expression syntax.
<code>\p</code>	The punctuation character <code>p</code> .
<code>\\</code>	The backslash character.
<code>\n</code>	The newline character <code>\u000A</code> .
<code>\t</code>	The tab character <code>\u0009</code> .
<code>\r</code>	The carriage return character <code>\u000D</code> .
<code>\f</code>	The form feed character <code>\u000C</code> .
<code>\e</code>	The escape character <code>\u001B</code> .
<code>\a</code>	The bell (alert) character <code>\u0007</code> .
<code>\uxxxx</code>	The Unicode character with hexadecimal code <code>xxxx</code> .
<code>\xxx</code>	The character with hexadecimal code <code>xx</code> .
<code>\On</code>	The character with octal code <code>n</code> .
<code>\Onn</code>	The character with octal code <code>nn</code> .
<code>\Onnn</code>	Character with octal code <code>nnn</code> , in which <code>nnn</code> $\leq$ 377.
<code>\cx</code>	The control character <code>^x</code> .
<b>Character classes</b>	
<code>[...]</code>	One of the characters between the brackets. Characters may be specified literally, and the syntax also allows the specification of character ranges, with intersection, union and subtraction operators. See specific examples that follow.
<code>[^...]</code>	Any one character not between the brackets.
<code>[a-z0-9]</code>	The character range: a character between (inclusive) <code>a</code> and <code>z</code> or <code>0</code> and <code>9</code> .
<code>[0-9[a-fA-F]]</code>	The union of classes: same as <code>[0-9a-fA-F]</code> .
<code>[a-z&amp;&amp;[aeiou]]</code>	The intersection of classes: same as <code>[aeiou]</code> .



Table 17–1. Java regular expression quick reference (continued)

Syntax	Matches
[a-z&&[^aeiou]]	Subtraction: the characters a through z, except for the vowels.
.	Any character, except a line terminator. If the DOTALL flag is set, it matches any character, including line terminators.
\d	An ASCII digit: [0-9].
\D	Anything but an ASCII digit: [^\d].
\s	ASCII whitespace: [ \t\n\f\r\x0B].
\S	Anything but ASCII whitespace: [^\s].
\w	An ASCII word character: [a-zA-Z0-9_].
\W	Anything but an ASCII word character: [^\w].
\p{group}	Any character in the named group. See the following group names. Many of the group names are from POSIX, which is why p is used for this character class.
\P{group}	Any character not in the named group.
\p{Lower}	An ASCII lowercase letter: [a-z].
\p{Upper}	An ASCII uppercase letter: [A-Z].
\p{ASCII}	Any ASCII character: [\x00-\x7f].
\p{Alpha}	An ASCII letter: [a-zA-Z].
\p{Digit}	An ASCII digit: [0-9]
\p{XDigit}	A hexadecimal digit: [0-9a-fA-F].
\p{Alnum}	ASCII letter or digit: [\p{Alpha}\p{Digit}].
\p{Punct}	ASCII punctuation: one of !"#\$%&'()*+,-./:;<=>?@[ \^_`{ }~].
\p{Graph}	A visible ASCII character: [\p{Alnum}\p{Punct}].
\p{Print}	A visible ASCII character: same as \p{Graph}.
\p{Blank}	An ASCII space or tab: [ \t].
\p{Space}	ASCII whitespace: [ \t\n\f\r\x0B].
\p{Cntrl}	An ASCII control character: [\x00-\x1f\x7f].
\p{category}	Any character in the named Unicode category. Category names are one- or two-letter codes defined by the Unicode standard. One-letter codes include L for letter, N for number, S for symbol, Z for separator, and P for punctuation. Two-letter codes represent subcategories, such as Lu for uppercase letter, Nd for decimal digit, Sc for currency symbol, Sm for math symbol, and Zs for space separator. See <code>java.lang.Character</code> for a set of constants that correspond to these subcategories, and note that the full set of one- and two-letter codes is not documented in this book.

## Pattern

Table 17–1. Java regular expression quick reference (continued)

Syntax	Matches
<code>\p{block}</code>	Any character in the named Unicode block. In Java regular expressions, block names begin with “In”, followed by mixed-case capitalization of the Unicode block name, without spaces or underscores. For example: <code>\p{InOgham}</code> or <code>\p{InMathematicalOperators}</code> . See <code>java.lang.Character.UnicodeBlock</code> for a list of Unicode block names.
<b>Sequences, alternatives, groups, and references</b>	
<code>xy</code>	Match <i>x</i> followed by <i>y</i> .
<code>x y</code>	Match <i>x</i> or <i>y</i> .
<code>(...)</code>	Grouping. Group subexpression within parentheses into a single unit that can be used with <code>*</code> , <code>+</code> , <code>?</code> , <code> </code> , and so on. Also “capture” the characters that match this group for later use.
<code>(?:...)</code>	Grouping only. Group subexpression as with <code>()</code> , but do not capture the text that matched.
<code>\n</code>	Match the same characters that were matched when capturing group number <i>n</i> was first matched. Be careful when <i>n</i> is followed by another digit: the largest number that is a valid group number will be used.
<b>Repetition<sup>a</sup></b>	
<code>x?</code>	Zero or one occurrence of <i>x</i> ; i.e., <i>x</i> is optional.
<code>x*</code>	Zero or more occurrences of <i>x</i> .
<code>x+</code>	One or more occurrences of <i>x</i> .
<code>x{n}</code>	Exactly <i>n</i> occurrences of <i>x</i> .
<code>x{n,}</code>	<i>n</i> or more occurrences of <i>x</i> .
<code>x{n,m}</code>	At least <i>n</i> , and at most <i>m</i> occurrences of <i>x</i> .
<b>Anchors<sup>b</sup></b>	
<code>^</code>	The beginning of the input string or, if the <code>MULTILINE</code> flag is specified, the beginning of the string or of any new line.
<code>\$</code>	The end of the input string or, if the <code>MULTILINE</code> flag is specified, the end of the string or of line within the string.
<code>\b</code>	A word boundary: a position in the string between a word and a non-word character.
<code>\B</code>	A position in the string that is not a word boundary.
<code>\A</code>	The beginning of the input string. Like <code>^</code> , but never matches the beginning of a new line, regardless of what flags are set.
<code>\Z</code>	The end of the input string, ignoring any trailing line terminator.

Table 17-1. Java regular expression quick reference (continued)

Syntax	Matches
<code>\z</code>	The end of the input string, including any line terminator.
<code>\G</code>	The end of the previous match.
<code>(?=x)</code>	A positive look-ahead assertion. Require that the following characters match <i>x</i> , but do not include those characters in the match.
<code>(?!x)</code>	A negative look-ahead assertion. Require that the following characters do not match the pattern <i>x</i> .
<code>(?&lt;=x)</code>	A positive look-behind assertion. Require that the characters immediately before the position match <i>x</i> , but do not include those characters in the match. <i>x</i> must be a pattern with a fixed number of characters.
<code>(?&lt;!x)</code>	A negative look-behind assertion. Require that the characters immediately before the position do not match <i>x</i> . <i>x</i> must be a pattern with a fixed number of characters.
<b>Miscellaneous</b>	
<code>(?&gt;x)</code>	Match <i>x</i> independently of the rest of the expression, without considering whether the match causes the rest of the expression to fail to match. Useful to optimize certain complex regular expressions. A group of this form does not capture the matched text.
<code>(?onflags-offflags)</code>	Don't match anything, but turn on the flags specified by <i>onflags</i> , and turn off the flags specified by <i>offflags</i> . These two strings are combinations in any order of the following letters and correspond to the following <b>Pattern</b> constants: <i>i</i> ( <b>CASE_INSENSITIVE</b> ), <i>d</i> ( <b>UNIX_LINES</b> ), <i>m</i> ( <b>MULTILINE</b> ), <i>s</i> ( <b>DOTALL</b> ), <i>u</i> ( <b>UNICODE_CASE</b> ), and <i>x</i> ( <b>COMMENTS</b> ). Flag settings specified in this way take effect at the point that they appear in the expression and persist until the end of the expression, or until the end of the parenthesized group of which they are a part, or until overridden by another flag setting expression.
<code>(?onflags-offflags:x)</code>	Match <i>x</i> , applying the specified flags to this subexpression only. This is a noncapturing group, such as <code>(?:...)</code> , with the addition of flags.
<code>\Q</code>	Don't match anything, but quote all subsequent pattern text until <code>\E</code> . All characters within such a quoted section are interpreted as literal characters to match, and none (except <code>\E</code> ) have special meanings.
<code>\E</code>	Don't match anything; terminate a quote started with <code>\Q</code> .

## Pattern

Table 17–1. Java regular expression quick reference (continued)

Syntax	Matches
<code>#comment</code>	If the <b>COMMENT</b> flag is set, pattern text between a <b>#</b> and the end of the line is considered a comment and is ignored.

<sup>a</sup> These repetition characters are known as greedy quantifiers because they match as many occurrences of *x* as possible while still allowing the rest of the regular expression to match. If you want a “reluctant quantifier,” which matches as few occurrences as possible while still allowing the rest of the regular expression to match, follow the previous quantifiers with a question mark. For example, use `*?` instead of `*`, and `{2,}?` instead of `{2,}`. Or, if you follow a quantifier with a plus sign instead of a question mark, then you specify a “possessive quantifier,” which matches as many occurrences as possible, even if it means that the rest of the regular expression will not match. Possessive quantifiers can be useful when you are sure that they will not adversely affect the rest of the match, because they can be implemented more efficiently than regular greedy quantifiers.

<sup>b</sup> Anchors do not match characters but instead match the zero-width positions between characters, “anchoring” the match to a position at which a specific condition holds.

Object	Pattern	Serializable
<pre> public final class <b>Pattern</b> implements Serializable { // No Constructor // Public Constants     public static final int <b>CANON_EQ</b>;                                =128     public static final int <b>CASE_INSENSITIVE</b>;                      =2     public static final int <b>COMMENTS</b>;                              =4     public static final int <b>DOTALL</b>;                                =32     public static final int <b>MULTILINE</b>;                              =8     public static final int <b>UNICODE_CASE</b>;                          =64     public static final int <b>UNIX_LINES</b>;                            =1 // Public Class Methods     public static Pattern <b>compile</b>(String regex);     public static Pattern <b>compile</b>(String regex, int flags);     public static boolean <b>matches</b>(String regex, CharSequence input); // Public Instance Methods     public int <b>flags</b>();     public Matcher <b>matcher</b>(CharSequence input);     public String <b>pattern</b>();     public String[] <b>split</b>(CharSequence input);     public String[] <b>split</b>(CharSequence input, int limit); } </pre>		

**Returned By:** `Matcher.pattern()`, `Pattern.compile()`

## PatternSyntaxException

Java 1.4

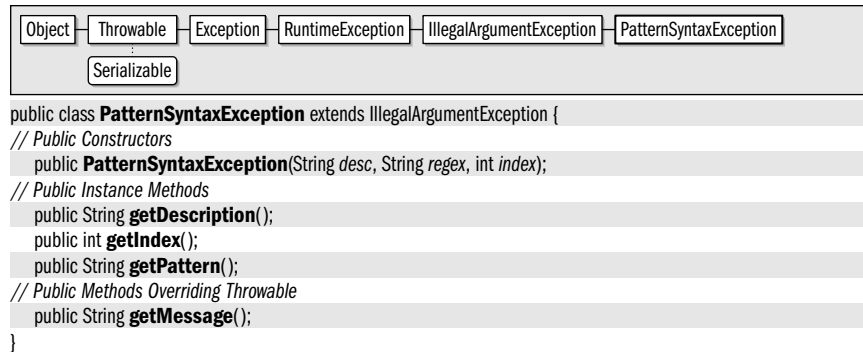
`java.util.regex`

*serializable unchecked*

This exception signals a syntax error in the text representation of a regular expression. An exception of this type may be thrown by the `Pattern.compile()` and `Pattern.matches()` methods, and also by the `String matches()`, `replaceFirst()`, `replaceAll()` and `split()` methods which call those `Pattern` methods.

`getPattern()` returns the text that contained the syntax error, and `getIndex()` returns the approximate location of the error within that text, or `-1`, if the location is not known.

getDescription() returns an error message that provides further detail about the error. The inherited getMessage() method combines the information provided by these other three methods into a single multiline message.



## Package java.util.zip

Java 1.1

The java.util.zip package contains classes for data compression and decompression. The Deflater and Inflater classes perform data compression and decompression. DeflaterOutputStream and InflaterInputStream apply that functionality to byte streams; the subclasses of these streams implement both the GZIP and ZIP compression formats. The Adler32 and CRC32 classes implement the Checksum interface and compute the checksums required for data compression.

### Interfaces:

```
public interface Checksum;
```

### Classes:

```

public class Adler32 implements Checksum;
public class CheckedInputStream extends java.io.FilterInputStream;
public class CheckedOutputStream extends java.io.FilterOutputStream;
public class CRC32 implements Checksum;
public class Deflater;
public class DeflaterOutputStream extends java.io.FilterOutputStream;
    public class GZIPOutputStream extends DeflaterOutputStream;
    public class ZipOutputStream extends DeflaterOutputStream;
public class Inflater;
public class InflaterInputStream extends java.io.FilterInputStream;
    public class GZIPInputStream extends InflaterInputStream;
    public class ZipInputStream extends InflaterInputStream;
public class ZipEntry implements Cloneable;
public class ZipFile;
  
```

### Exceptions:

```

public class DataFormatException extends Exception;
public class ZipException extends java.io.IOException;
  
```

**Adler32**

Java 1.1

java.util.zip

This class implements the `Checksum` interface and computes a checksum on a stream of data using the Adler-32 algorithm. This algorithm is significantly faster than the CRC-32 algorithm and is almost as reliable. The `CheckedInputStream` and `CheckedOutputStream` classes provide a higher-level interface to computing checksums on streams of data.

```

Object — Adler32 — Checksum

public class Adler32 implements Checksum {
    // Public Constructors
    public Adler32();
    // Public Instance Methods
    public void update(byte[] b);
    // Methods Implementing Checksum
    public long getValue(); default:1
    public void reset();
    public void update(int b);
    public void update(byte[] b, int off, int len);
}

```

**CheckedInputStream**

Java 1.1

java.util.zip

This class is a subclass of `java.io.FilterInputStream`; it allows a stream to be read and a checksum computed on its contents at the same time. This is useful when you want to check the integrity of a stream of data against a published checksum value. To create a `CheckedInputStream`, you must specify both the stream it should read and a `Checksum` object, such as `CRC32`, that implements the particular checksum algorithm you desire. The `read()` and `skip()` methods are the same as those of other input streams. As bytes are read, they are incorporated into the checksum that is being computed. The `getChecksum()` method does not return the checksum value itself, but rather the `Checksum` object. You must call the `getValue()` method of this object to obtain the checksum value.

```

Object — InputStream — FilterInputStream — CheckedInputStream

public class CheckedInputStream extends java.io.FilterInputStream {
    // Public Constructors
    public CheckedInputStream(java.io.InputStream in, Checksum cksum);
    // Public Instance Methods
    public Checksum getChecksum();
    // Public Methods Overriding FilterInputStream
    public int read() throws java.io.IOException;
    public int read(byte[] buf, int off, int len) throws java.io.IOException;
    public long skip(long n) throws java.io.IOException;
}

```

**CheckedOutputStream**

Java 1.1

java.util.zip

This class is a subclass of `java.io.FilterOutputStream` that allows data to be written to a stream and a checksum computed on that data at the same time. To create a `CheckedOutputStream`, you must specify both the output stream to write its data to and a `Checksum` object, such as an instance of `Adler32`, that implements the particular checksum algorithm you desire. The `write()` methods are similar to those of other `OutputStream` classes.

The `getChecksum()` method returns the `Checksum` object. You must call `getValue()` on this object in order to obtain the actual checksum value.

```

Object -- OutputStream -- FilterOutputStream -- CheckedOutputStream

public class CheckedOutputStream extends java.io.FilterOutputStream {
// Public Constructors
    public CheckedOutputStream(java.io.OutputStream out, Checksum cksum);
// Public Instance Methods
    public Checksum getChecksum();
// Public Methods Overriding FilterOutputStream
    public void write(int b) throws java.io.IOException;
    public void write(byte[] b, int off, int len) throws java.io.IOException;
}

```

## Checksum

Java 1.1

java.util.zip

This interface defines the methods required to compute a checksum on a stream of data. The checksum is computed based on the bytes of data supplied by the `update()` methods; the current value of the checksum can be obtained at any time with the `getValue()` method. `reset()` resets the checksum to its default value; use this method before beginning a new stream of data. The checksum value computed by a `Checksum` object and returned through the `getValue()` method must fit into a `long` value. Therefore, this interface is not suitable for the cryptographic checksum algorithms used in cryptography and security. The classes `CheckedInputStream` and `CheckedOutputStream` provide a higher-level API for computing a checksum on a stream of data. See also `java.security.MessageDigest`.

```

public interface Checksum {
// Public Instance Methods
    public abstract long getValue();
    public abstract void reset();
    public abstract void update(int b);
    public abstract void update(byte[] b, int off, int len);
}

```

*Implementations:* Adler32, CRC32

*Passed To:* `CheckedInputStream.CheckedInputStream()`,  
`CheckedOutputStream.CheckedOutputStream()`

*Returned By:* `CheckedInputStream.getChecksum()`, `CheckedOutputStream.getChecksum()`

## CRC32

Java 1.1

java.util.zip

This class implements the `Checksum` interface and computes a checksum on a stream of data using the CRC-32 algorithm. The `CheckedInputStream` and `CheckedOutputStream` classes provide a higher-level interface to computing checksums on streams of data.

```

Object -- CRC32 -- Checksum

public class CRC32 implements Checksum {
// Public Constructors
    public CRC32();
// Public Instance Methods
    public void update(byte[] b);
}

```

```
// Methods Implementing Checksum
public long getValue();                                default:0
public void reset();
public void update(int b);
public void update(byte[] b, int off, int len);
}
```

Type Of: GZIPInputStream.crc, GZIPOutputStream.crc

## DataFormatException

Java 1.1

java.util.zip

serializable checked

Signals that invalid or corrupt data has been encountered while uncompressing data.



```
public class DataFormatException extends Exception {
// Public Constructors
public DataFormatException();
public DataFormatException(String s);
}
```

Thrown By: Inflater.inflate()

## Deflater

Java 1.1

java.util.zip

This class implements the general ZLIB data-compression algorithm used by the *gzip* and *PKZip* compression programs. The constants defined by this class are used to specify the compression strategy and the compression speed/strength tradeoff level to be used. If you set the *nowrap* argument to the constructor to **true**, the ZLIB header and checksum data are omitted from the compressed output, which is the format both *gzip* and *PKZip* use.

The important methods of this class are **setInput()**, which specifies input data to be compressed, and **deflate()**, which compresses the data and returns the compressed output. The remaining methods exist so that **Deflater** can be used for stream-based compression, as it is in higher-level classes, such as **GZIPOutputStream** and **ZipOutputStream**. These stream classes are sufficient in most cases. Most applications do not need to use **Deflater** directly. The **Inflater** class uncompresses data compressed with a **Deflater** object.

```
public class Deflater {
// Public Constructors
public Deflater();
public Deflater(int level);
public Deflater(int level, boolean nowrap);
// Public Constants
public static final int BEST_COMPRESSION;           =9
public static final int BEST_SPEED;                 =1
public static final int DEFAULT_COMPRESSION;       =-1
public static final int DEFAULT_STRATEGY;          =0
public static final int DEFLATED;                  =8
public static final int FILTERED;                  =1
public static final int HUFFMAN_ONLY;               =2
public static final int NO_COMPRESSION;            =0
}
```



```
// Property Accessor Methods (by property name)
public int getAdler();                                synchronized default:1
public int getTotalIn();                                synchronized default:0
public int getTotalOut();                                synchronized default:0
// Public Instance Methods
public int deflate(byte[] b);
public int deflate(byte[] b, int off, int len);          synchronized
public void end();                                        synchronized
public void finish();                                    synchronized
public boolean finished();                                synchronized
public boolean needsInput();
public void reset();                                    synchronized
public void setDictionary(byte[] b);
public void setDictionary(byte[] b, int off, int len);    synchronized
public void setInput(byte[] b);
public void setInput(byte[] b, int off, int len);          synchronized
public void setLevel(int level);                        synchronized
public void setStrategy(int strategy);                  synchronized
// Protected Methods Overriding Object
protected void finalize();
}
```

*Passed To:* DeflaterOutputStream.DeflaterOutputStream()

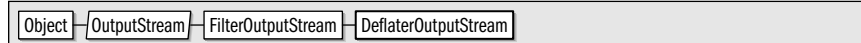
*Type Of:* DeflaterOutputStream.def

## DeflaterOutputStream

Java 1.1

java.util.zip

This class is a subclass of `java.io.FilterOutputStream`; it filters a stream of data by compressing (deflating) it and then writing the compressed data to another output stream. To create a `DeflaterOutputStream`, you must specify both the stream it is to write to and a `Deflater` object to perform the compression. You can set various options on the `Deflater` object to specify just what type of compression is to be performed. Once a `DeflaterOutputStream` is created, its `write()` and `close()` methods are the same as those of other output streams. The `InflaterInputStream` class can read data written with a `DeflaterOutputStream`. A `DeflaterOutputStream` writes raw compressed data; applications often prefer one of its subclasses, `GZIPOutputStream` or `ZipOutputStream`, that wraps the raw compressed data within a standard file format.



```
public class DeflaterOutputStream extends java.io.FilterOutputStream {
// Public Constructors
public DeflaterOutputStream(java.io.OutputStream out);
public DeflaterOutputStream(java.io.OutputStream out, Deflater def);
public DeflaterOutputStream(java.io.OutputStream out, Deflater def, int size);
// Public Instance Methods
public void finish() throws java.io.IOException;
// Public Methods Overriding FilterOutputStream
public void close() throws java.io.IOException;
public void write(int b) throws java.io.IOException;
public void write(byte[] b, int off, int len) throws java.io.IOException;
// Protected Instance Methods
protected void deflate() throws java.io.IOException;
// Protected Instance Fields
}
```

## *DeflaterOutputStream*

```
protected byte[] buf;  
protected Deflater def;  
}
```

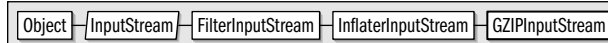
*Subclasses:* GZIPOutputStream, ZipOutputStream

## **GZIPInputStream**

**Java 1.1**

java.util.zip

This class is a subclass of *InflaterInputStream* that reads and uncompresses data compressed in *gzip* format. To create a *GZIPInputStream*, simply specify the *InputStream* to read compressed data from and, optionally, a buffer size for the internal decompression buffer. Once a *GZIPInputStream* is created, you can use the *read()* and *close()* methods as you would with any input stream.



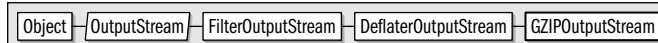
```
public class GZIPInputStream extends InflaterInputStream {  
    // Public Constructors  
    public GZIPInputStream(java.io.InputStream in) throws java.io.IOException;  
    public GZIPInputStream(java.io.InputStream in, int size) throws java.io.IOException;  
    // Public Constants  
    public static final int GZIP_MAGIC;                                     =35615  
    // Public Methods Overriding InflaterInputStream  
    public void close() throws java.io.IOException;  
    public int read(byte[] buf, int off, int len) throws java.io.IOException;  
    // Protected Instance Fields  
    protected CRC32 crc;  
    protected boolean eos;  
}
```

## **GZIPOutputStream**

**Java 1.1**

java.util.zip

This class is a subclass of *DeflaterOutputStream* that compresses and writes data using the *gzip* file format. To create a *GZIPOutputStream*, specify the *OutputStream* to write to and, optionally, a size for the internal compression buffer. Once the *GZIPOutputStream* is created, you can use the *write()* and *close()* methods as you would any output stream.



```
public class GZIPOutputStream extends DeflaterOutputStream {  
    // Public Constructors  
    public GZIPOutputStream(java.io.OutputStream out) throws java.io.IOException;  
    public GZIPOutputStream(java.io.OutputStream out, int size) throws java.io.IOException;  
    // Public Methods Overriding DeflaterOutputStream  
    public void finish() throws java.io.IOException;  
    public void write(byte[] buf, int off, int len) throws java.io.IOException;    synchronized  
    // Protected Instance Fields  
    protected CRC32 crc;  
}
```

**Inflater****Java 1.1****java.util.zip**

This class implements the general ZLIB data-decompression algorithm used by *gzip*, *PKZip*, and other data-compression applications. It decompresses or inflates data compressed through the `Deflater` class. The important methods of this class are `setInput()`, which specifies input data to be decompressed, and `inflate()`, which decompresses the input data into an output buffer. A number of other methods exist so that this class can be used for stream-based decompression, as it is in the higher-level classes, such as `GZIPInputStream` and `ZipInputStream`. These stream-based classes are sufficient in most cases. Most applications do not need to use `Inflater` directly.

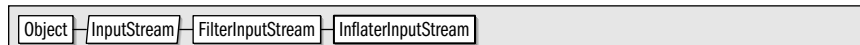
```
public class Inflater {
// Public Constructors
    public Inflater();
    public Inflater(boolean nowrap);
// Property Accessor Methods (by property name)
    public int getAdler();                                synchronized default:1
    public int getRemaining();                            synchronized default:0
    public int getTotalIn();                              synchronized default:0
    public int getTotalOut();                             synchronized default:0
// Public Instance Methods
    public void end();                                    synchronized
    public boolean finished();                            synchronized
    public int inflate(byte[] b) throws DataFormatException;
    public int inflate(byte[] b, int off, int len) throws DataFormatException;    synchronized
    public boolean needsDictionary();                    synchronized
    public boolean needsInput();                        synchronized
    public void reset();                                synchronized
    public void setDictionary(byte[] b);
    public void setDictionary(byte[] b, int off, int len);    synchronized
    public void setInput(byte[] b);
    public void setInput(byte[] b, int off, int len);        synchronized
// Protected Methods Overriding Object
    protected void finalize();
}
```

*Passed To:* `InflaterInputStream.InflaterInputStream()`

*Type Of:* `InflaterInputStream.inf`

**InflaterInputStream****Java 1.1****java.util.zip**

This class is a subclass of `java.io.FilterInputStream`; it reads a specified stream of compressed input data (typically, one that was written with `DeflaterOutputStream` or a subclass) and filters that data by uncompressing (inflating) it. To create an `InflaterInputStream`, specify both the input stream to read from and an `Inflater` object to perform the decompression. Once an `InflaterInputStream` is created, the `read()` and `skip()` methods are the same as those of other input streams. The `InflaterInputStream` uncompresses raw data. Applications often prefer one of its subclasses, `GZIPInputStream` or `ZipInputStream`, that work with compressed data written in the standard *gzip* and *PKZip* file formats.



## *InflaterInputStream*

```
public class InflaterInputStream extends java.io.FilterInputStream {
// Public Constructors
    public InflaterInputStream(java.io.InputStream in);
    public InflaterInputStream(java.io.InputStream in, Inflater inf);
    public InflaterInputStream(java.io.InputStream in, Inflater inf, int size);
// Public Methods Overriding FilterInputStream
    1.2 public int available() throws java.io.IOException;
    1.2 public void close() throws java.io.IOException;
    public int read() throws java.io.IOException;
    public int read(byte[] b, int off, int len) throws java.io.IOException;
    public long skip(long n) throws java.io.IOException;
// Protected Instance Methods
    protected void fill() throws java.io.IOException;
// Protected Instance Fields
    protected byte[] buf;
    protected Inflater inf;
    protected int len;
}
```

*Subclasses:* GZIPInputStream, ZipInputStream

## **ZipEntry**

**Java 1.1**

**java.util.zip**

*cloneable*

This class describes a single entry (typically a compressed file) stored within a ZIP file. The various methods get and set various pieces of information about the entry. The ZipEntry class is used by ZipFile and ZipInputStream, which read ZIP files, and by ZipOutputStream, which writes ZIP files.

When you are reading a ZIP file, a ZipEntry object returned by ZipFile or ZipInputStream contains the name, size, modification time, and other information about an entry in the file. When writing a ZIP file, on the other hand, you must create your own ZipEntry objects and initialize them to contain the entry name and other appropriate information before writing the contents of the entry.



```
public class ZipEntry implements Cloneable {
// Public Constructors
    public ZipEntry(String name);
    1.2 public ZipEntry(ZipEntry e);
// Public Constants
    public static final int DEFLATED;           =8
    public static final int STORED;             =0
// Property Accessor Methods (by property name)
    public String getComment();
    public void setComment(String comment);
    public long getCompressedSize();
    1.2 public void setCompressedSize(long csize);
    public long getCrc();
    public void setCrc(long crc);
    public boolean isDirectory();
    public byte[] getExtra();
    public void setExtra(byte[] extra);
}
```

```

    public int getMethod();
    public void setMethod(int method);
    public String getName();
    public long getSize();
    public void setSize(long size);
    public long getTime();
    public void setTime(long time);
    // Public Methods Overriding Object
    1.2 public Object clone();
    1.2 public int hashCode();
    public String toString();
}

```

*Subclasses:* java.util.jar.JarEntry

*Passed To:* java.util.jar.JarEntry.JarEntry(), java.util.jar.JarFile.getInputStream(),  
java.util.jar.JarOutputStream.putNextEntry(), ZipEntry.ZipEntry(), ZipFile.getInputStream(),  
ZipOutputStream.putNextEntry()

*Returned By:* java.util.jar.JarFile.getEntry(), java.util.jar.JarInputStream.{createZipEntry(),  
getNextEntry()}, ZipFile.getEntry(), ZipInputStream.{createZipEntry(), getNextEntry()}

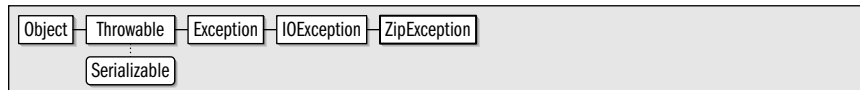
## ZipException

Java 1.1

java.util.zip

*serializable checked*

Signals that an error has occurred in reading or writing a ZIP file.



```

public class ZipException extends java.io.IOException {
    // Public Constructors
    public ZipException();
    public ZipException(String s);
}

```

*Subclasses:* java.util.jar.JarException

*Thrown By:* ZipFile.ZipFile()

## ZipFile

Java 1.1

java.util.zip

This class reads the contents of ZIP files. It uses a random-access file internally so that the entries of the ZIP file do not have to be read sequentially, as they do with the ZipInputStream class. A ZipFile object can be created by specifying the ZIP file to be read either as a String filename or as a File object. In Java 1.3, temporary ZIP files can be marked for automatic deletion when they are closed. To take advantage of this feature, pass ZipFile.OPEN\_READ|ZipFile.OPEN\_DELETE as the *mode* argument to the ZipFile() constructor.

Once a ZipFile is created, the getEntry() method returns a ZipEntry object for a named entry, and the entries() method returns an Enumeration object that allows you to loop through all the ZipEntry objects for the file. To read the contents of a specific ZipEntry

## ZipFile

within the ZIP file, pass the `ZipEntry` to `getInputStream()`; this returns an `InputStream` object from which you can read the entry's contents.

Object — ZipFile — ZipConstants

```
public class ZipFile {  
    // Public Constructors  
    public ZipFile(String name) throws java.io.IOException;  
    public ZipFile(java.io.File file) throws ZipException, java.io.IOException;  
    1.3 public ZipFile(java.io.File file, int mode) throws java.io.IOException;  
    // Public Constants  
    1.3 public static final int OPEN_DELETE; =4  
    1.3 public static final int OPEN_READ; =1  
    // Public Instance Methods  
    public void close() throws java.io.IOException;  
    public java.util.Enumeration entries();  
    public ZipEntry getEntry(String name);  
    public java.io.InputStream getInputStream(ZipEntry entry) throws java.io.IOException;  
    public String getName();  
    1.2 public int size();  
    // Protected Methods Overriding Object  
    1.3 protected void finalize() throws java.io.IOException;  
}
```

*Subclasses:* `java.util.jar.JarFile`

## ZipInputStream

Java 1.1

`java.util.zip`

This class is a subclass of `InflaterInputStream` that reads the entries of a ZIP file in sequential order. Create a `ZipInputStream` by specifying the `InputStream` from which it is to read the contents of the ZIP file. Once the `ZipInputStream` is created, you can use `getNextEntry()` to begin reading data from the next entry in the ZIP file. This method must be called before `read()` is called to begin reading the first entry. `getNextEntry()` returns a `ZipEntry` object that describes the entry being read, or null when there are no more entries to be read from the ZIP file.

The `read()` methods of `ZipInputStream` read until the end of the current entry and then return `-1`, indicating that there is no more data to read. To continue with the next entry in the ZIP file, you must call `getNextEntry()` again. Similarly, the `skip()` method only skips bytes within the current entry. `closeEntry()` can be called to skip the remaining data in the current entry, but it is usually easier simply to call `getNextEntry()` to begin the next entry.

Object — InputStream — FilterInputStream — InflaterInputStream — ZipInputStream — ZipConstants

```
public class ZipInputStream extends InflaterInputStream {  
    // Public Constructors  
    public ZipInputStream(java.io.InputStream in);  
    // Public Instance Methods  
    public void closeEntry() throws java.io.IOException;  
    public ZipEntry getNextEntry() throws java.io.IOException;  
    // Public Methods Overriding InflaterInputStream  
    1.2 public int available() throws java.io.IOException;  
    public void close() throws java.io.IOException;  
}
```

```

    public int read(byte[] b, int off, int len) throws java.io.IOException;
    public long skip(long n) throws java.io.IOException;
    // Protected Instance Methods
    1.2 protected ZipEntry createZipEntry(String name);
}

```

*Subclasses:* java.util.jar.JarInputStream

## ZipOutputStream

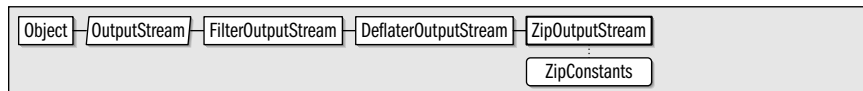
Java 1.1

java.util.zip

This class is a subclass of `DeflaterOutputStream` that writes data in ZIP file format to an output stream. Before writing any data to the `ZipOutputStream`, you must begin an entry within the ZIP file with `putNextEntry()`. The `ZipEntry` object passed to this method should specify at least a name for the entry. Once you have begun an entry with `putNextEntry()`, you can write the contents of that entry with the `write()` methods. When you reach the end of an entry, you can begin a new one by calling `putNextEntry()` again, you can close the current entry with `closeEntry()`, or you can close the stream itself with `close()`.

Before beginning an entry with `putNextEntry()`, you can set the compression method and level with `setMethod()` and `setLevel()`. The constants `DEFLATED` and `STORED` are the two legal values for `setMethod()`. If you use `STORED`, the entry is stored in the ZIP file without any compression. If you use `DEFLATED`, you can also specify the compression speed/strength tradeoff by passing a number from 1 to 9 to `setLevel()`, in which 9 gives the strongest and slowest level of compression. You can also use the constants `Deflater.BEST_SPEED`, `Deflater.BEST_COMPRESSION`, and `Deflater.DEFAULT_COMPRESSION` with the `setLevel()` method.

If you are storing an entry without compression, the ZIP file format requires that you specify, in advance, the entry size and CRC-32 checksum in the `ZipEntry` object for the entry. An exception is thrown if these values are not specified or specified incorrectly.



```

public class ZipOutputStream extends DeflaterOutputStream {
    // Public Constructors
    public ZipOutputStream(java.io.OutputStream out);
    // Public Constants
    public static final int DEFLATED;           =8
    public static final int STORED;             =0
    // Public Instance Methods
    public void closeEntry() throws java.io.IOException;
    public void putNextEntry(ZipEntry e) throws java.io.IOException;
    public void setComment(String comment);
    public void setLevel(int level);
    public void setMethod(int method);
    // Public Methods Overriding DeflaterOutputStream
    public void close() throws java.io.IOException;
    public void finish() throws java.io.IOException;
    public void write(byte[] b, int off, int len) throws java.io.IOException;    synchronized
}

```

*Subclasses:* java.util.jar.JarOutputStream