Table of Contents

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Chapter 16. 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 important and commonly used utility classes, the most important of which are the various `Collection`, `Set`, `List`, and `Map` implementations. In Java 5.0 the collection classes and interfaces have been converted into generic types.

`java.util.concurrent`

This package includes utilities for concurrent programming, including threadsafe collection classes, threadpool implementations, and synchronizer utilities.

`java.util.concurrent.atomic`

This package includes classes that define atomic operations on primitive values or object references.

`java.util.concurrent.locks`

This package contains low-level lock and condition utilities.

`java.util.jar`

This package defines classes for reading and writing JAR (Java ARchive) files. They 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 in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`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 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. Java 5.0 adds a `Queue` collection interface and implementations. It also adds `EnumSet` and `EnumMap` which efficiently implement the `Set` and `Map` interfaces for use with enumerated types. Most importantly, Java 5.0 modifies all collection interfaces and classes to be generic types, which enable type-safe collections such as `List<String>`. `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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

read in by an internationalized program at runtime. `Random` generates and returns pseudorandom numbers in a variety of forms. `StringTokenizer` is a simple 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. In Java 5.0, the `Formatter` class enables poweful formatted text output in the style of the C programming language's `printf( )` function. The Java 5.0 `Scanner` class is a text tokenizer or scanner that can also parse numbers and match tokens based on regular expressions.

Interfaces

```
public interface Collection<E> extends Iterable<E>;
public interface Comparator<T>;
public interface Enumeration<E>;
public interface EventListener;
public interface Formattable;
public interface Iterator<E>;
public interface List<E> extends Collection<E>;
public interface ListIterator<E> extends Iterator<E>;
public interface Map<K, V>;
public interface Map.Entry<K, V>;
public interface Observer;
public interface Queue<E> extends Collection<E>;
public interface RandomAccess;
public interface Set<E> extends Collection<E>;
public interface SortedMap<K, V> extends Map<K, V>;
public interface SortedSet<E> extends Set<E>;
```

Enumerated Types

```
public enum Formatter.BigDecimalLayoutForm;
```

Collections

```
public abstract class AbstractCollection<E> implements Collection<E>;
   public abstract class AbstractList<E> extends AbstractCollection<E>
        implements List<E>;
      public abstract class AbstractSequentialList<E> extends AbstractList<E>;
         public class LinkedList<E> extends AbstractSequentialList<E>
                     implements List<E>, Queue<E>, Cloneable, Serializable;
      public class ArrayList<E> extends AbstractList<E> implements List<E>,
        RandomAccess, Cloneable, Serializable;
      public class Vector<E> extends AbstractList<E> implements List<E>,
        RandomAccess, Cloneable, Serializable;
         public class Stack<E> extends Vector<E>;
   public abstract class AbstractQueue<E> extends AbstractCollection<E>
        implements Queue<E>;
      public class PriorityQueue<E> extends AbstractQueue<E>
        implements Serializable;
   public abstract class AbstractSet<E> extends AbstractCollection<E>
        implements Set<E>;
      public abstract class EnumSet<E extends Enum<E>> extends AbstractSet<E>
        implements Cloneable, Serializable;
      public class HashSet<E> extends AbstractSet<E> implements Set<E>,
        Cloneable, Serializable;
         public class LinkedHashSet<E> extends HashSet<E> implements Set<E>,
        Cloneable, Serializable;
      public class TreeSet<E> extends AbstractSet<E> implements SortedSet<E>,
        Cloneable, Serializable;
public abstract class AbstractMap<K, V> implements Map<K, V>;
   public class EnumMap<K extends Enum<K>, V> extends AbstractMap<K, V>
        implements Serializable, Cloneable;
   public class HashMap<K, V> extends AbstractMap<K, V> implements Map<K, V>,
        Cloneable, Serializable;
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
      public class LinkedHashMap<K, V> extends HashMap<K, V>
        implements Map<K, V>;
   public class IdentityHashMap<K, V> extends AbstractMap<K, V>
        implements Map<K, V>, Serializable, Cloneable;
   public class TreeMap<K, V> extends AbstractMap<K, V>
        implements SortedMap<K, V>, Cloneable, Serializable;
   public class WeakHashMap<K, V> extends AbstractMap<K, V> implements Map<K, V>;
public class Hashtable<K, V> extends Dictionary<K, V> implements Map<K, V>,
        Cloneable, Serializable;
   public class Properties extends Hashtable<Object, Object>;
```

Events

```
public class EventObject implements Serializable;
```

Other Classes

```
public class Arrays;
public class BitSet implements Cloneable, Serializable;
public abstract class Calendar implements Serializable, Cloneable, Comparable<Calendar>;
   public class GregorianCalendar extends Calendar;
public class Collections;
public final class Currency implements Serializable;
public class Date implements Serializable, Cloneable, Comparable<Date>;
public abstract class Dictionary<K, V>;
public abstract class EventListenerProxy implements EventListener;
public class FormattableFlags;
public final class Formatter implements java.io.Closeable, java.io.Flushable;
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 final class Scanner implements Iterator<String>;
public class StringTokenizer implements Enumeration<Object>;
public class Timer;
public abstract class TimerTask implements Runnable;
public abstract class TimeZone implements Cloneable, Serializable;
   public class SimpleTimeZone extends TimeZone;
public final class UUID implements Serializable, Comparable<UUID>;
```

Exceptions

```
public class ConcurrentModificationException extends RuntimeException;
public class EmptyStackException extends RuntimeException;
public class FormatterClosedException extends IllegalStateException;
public class IllegalFormatException extends IllegalArgumentException;
   public class DuplicateFormatFlagsException extends IllegalFormatException;
   public class FormatFlagsConversionMismatchException extends IllegalFormatException;
   public class IllegalFormatCodePointException extends IllegalFormatException;
   public class IllegalFormatConversionException extends IllegalFormatException;
   public class IllegalFormatFlagsException extends IllegalFormatException;
   public class IllegalFormatPrecisionException extends IllegalFormatException;
   public class IllegalFormatWidthException extends IllegalFormatException;
   public class MissingFormatArgumentException extends IllegalFormatException;
   public class MissingFormatWidthException extends IllegalFormatException;
   public class UnknownFormatConversionException extends IllegalFormatException;
   public class UnknownFormatFlagsException extends IllegalFormatException;
public class InvalidPropertiesFormatException extends java.io.IOException;
public class MissingResourceException extends RuntimeException;
public class NoSuchElementException extends RuntimeException;
   public class InputMismatchException extends NoSuchElementException;
public class TooManyListenersException extends Exception;
```

AbstractCollection<E> java.util

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.2 ***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 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`.

Figure 16-1. java.util.AbstractCollection<E>


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

Subclasses

`AbstractList`, `AbstractQueue`, `AbstractSet`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

AbstractList<E> **java.util**

Java 1.2 ***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.

Figure 16-2. java.util.AbstractList<E>


```
public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E> {
// Protected Constructors
     protected AbstractList( );
// Methods Implementing List
     public boolean add(E o);
     public void add(int index, E element);
     public boolean addAll(int index, Collection<? extends E> c);
     public void clear( );
     public boolean equals(Object o);
     public abstract E get(int index);
     public int hashCode( );
     public int indexOf(Object o);
     public Iterator<E> iterator( );
     public int lastIndexOf(Object o);
     public ListIterator<E> listIterator( );
     public ListIterator<E> listIterator(int index);
     public E remove(int index);
     public E set(int index, E element);
     public List<E> subList(int fromIndex, int toIndex);
// Protected Instance Methods
     protected void removeRange(int fromIndex, int toIndex);
// Protected Instance Fields
     protected transient int modCount;
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Subclasses

`AbstractSequentialList`, `ArrayList`, `Vector`

AbstractMap<K,V> java.util

Java 1.2 ***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 are are substantially more efficient.

Figure 16-3. java.util.AbstractMap<K,V>


```
public abstract class AbstractMap<K,V> implements Map<K,V> {
// Protected Constructors
     protected AbstractMap( );
// Methods Implementing Map
     public void clear( );
     public boolean containsKey(Object key);
     public boolean containsValue(Object value);
     public abstract Set<Map.Entry<K,V>> entrySet( );
     public boolean equals(Object o);
     public V get(Object key);
     public int hashCode( );
     public boolean isEmpty( );
     public Set<K> keySet( );
     public V put(K key, V value);
     public void putAll(Map<? extends K,? extends V> t);
     public V remove(Object key);
     public int size( );
     public Collection<V> values( );
// Public Methods Overriding Object
     public String toString( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
// Protected Methods Overriding Object
1.4  protected Object clone( ) throws CloneNotSupportedException;
}
```

Subclasses

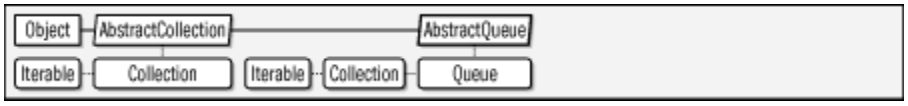
`EnumMap`, `HashMap`, `IdentityHashMap`, `TreeMap`, `WeakHashMap`, `java.util.concurrent.ConcurrentHashMap`

AbstractQueue<E> java.util

Java 5.0 ***collection***

This abstract class provides a framework for simple `Queue` implementations. A concrete subclass must implement `offer( )`, `peek( )`, and `poll( )` and must also implement the inherited `size( )` and `iterator( )` methods of the `Collection` interface. The `Iterator` returned by `iterator( )` must support the `remove( )` operation.

Figure 16-4. java.util.AbstractQueue<E>




```
public abstract class AbstractQueue<E> extends AbstractCollection<E> implements Queue<E> {
// Protected Constructors
     protected AbstractQueue( );
// Methods Implementing Collection
     public boolean add(E o);
     public boolean addAll(Collection<? extends E> c);
     public void clear( );
// Methods Implementing Queue
     public E element( );
     public E remove( );
}
```

Subclasses

`PriorityQueue`, `java.util.concurrent.ArrayBlockingQueue`, `java.util.concurrent.ConcurrentLinkedQueue`, `java.util.concurrent.DelayQueue`, `java.util.concurrent.LinkedBlockingQueue`, `java.util.concurrent.PriorityBlockingQueue`, `java.util.concurrent.SynchronousQueue`

AbstractSequentialList<E> java.util

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.2 ***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.

Figure 16-5. java.util.AbstractSequentialList<E>


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

Subclasses

`LinkedList`

AbstractSet<E> **java.util**

Java 1.2 ***collection***

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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`.

Figure 16-6. java.util.AbstractSet<E>


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

Subclasses

`EnumSet`, `HashSet`, `TreeSet`, `java.util.concurrent.CopyOnWriteArraySet`

ArrayList<E> java.util

Java 1.2 ***cloneable serializable collection***

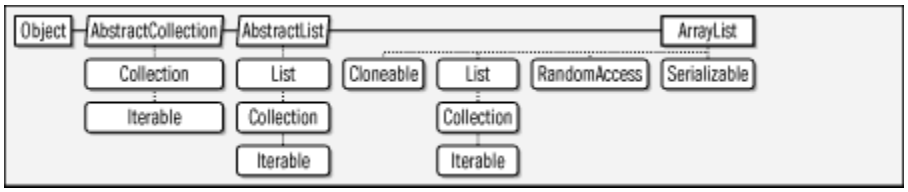
This class is a `List` implementation based on an array (that is recreated as necessary as the list grows or shrinks). `ArrayList` implements all optional `List` and `Collection` methods and allows list elements of any type (including `null`). Because `ArrayList` is based on an array, the `get( )` and `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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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.

Figure 16-7. java.util.ArrayList<E>




```
public class ArrayList<E> extends AbstractList<E> implements List<E>,
        RandomAccess, Cloneable, Serializable {
// Public Constructors
     public ArrayList( );
     public ArrayList(int initialCapacity);
     public ArrayList(Collection<? extends E> c);
// Public Instance Methods
     public void ensureCapacity(int minCapacity);
     public void trimToSize( );
// Methods Implementing List
     public boolean add(E o);
     public void add(int index, E element);
     public boolean addAll(Collection<? extends E> c);
     public boolean addAll(int index, Collection<? extends E> c);
     public void clear( );
     public boolean contains(Object elem);
     public E get(int index);
     public int indexOf(Object elem);
     public boolean isEmpty( );                              default:true
     public int lastIndexOf(Object elem);
5.0  public boolean remove(Object o);
     public E remove(int index);
     public E set(int index, E element);
     public int size( );
     public Object[ ] toArray( );
     public <T> T[ ] toArray(T[ ] a);
// Protected Methods Overriding AbstractList
     protected void removeRange(int fromIndex, int toIndex);
// Public Methods Overriding Object
     public Object clone( );
}
```

Returned By

`Collections.list( )`

Arrays **java.util**

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.2

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.

Java 5.0 adds `hashCode( )` methods that compute a hashcode for the contents of the array. These methods are compatible with the `equals( )` methods: `equal( )` arrays will always have the same `hashCode( )`. Java 5.0 also adds `deepEquals( )` and `deepHashCode( )` methods that handle multi-dimensional arrays. Finally, the Java 5.0 `toString( )` and `deepToString( )` methods convert arrays to strings. The returned strings are a comma-separated list of elements enclosed in square brackets.

```
public class Arrays {
// No Constructor
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
// Public Class Methods
     public static <T> List<T> asList(T ... a);
     public static int binarySearch(char[ ] a, char key);
     public static int binarySearch(short[ ] a, short key);
     public static int binarySearch(long[ ] a, long key);
     public static int binarySearch(int[ ] a, int key);
     public static int binarySearch(float[ ] a, float key);
     public static int binarySearch(Object[ ] a, Object key);
     public static int binarySearch(byte[ ] a, byte key);
     public static int binarySearch(double[ ] a, double key);
     public static <T> int binarySearch(T[ ] a, T key, Comparator<? super T> c);
5.0  public static boolean deepEquals(Object[ ] a1, Object[ ] a2);
5.0  public static int deepHashCode(Object[ ] a);
5.0  public static String deepToString(Object[ ] a);
     public static boolean equals(boolean[ ] a, boolean[ ] a2);
     public static boolean equals(long[ ] a, long[ ] a2);
     public static boolean equals(float[ ] a, float[ ] a2);
     public static boolean equals(double[ ] a, double[ ] a2);
     public static boolean equals(char[ ] a, char[ ] a2);
     public static boolean equals(byte[ ] a, byte[ ] a2);
     public static boolean equals(int[ ] a, int[ ] a2);
     public static boolean equals(short[ ] a, short[ ] a2);
     public static boolean equals(Object[ ] a, Object[ ] a2);
     public static void fill(char[ ] a, char val);
     public static void fill(short[ ] a, short val);
     public static void fill(byte[ ] a, byte val);
     public static void fill(int[ ] a, int val);
     public static void fill(double[ ] a, double val);
     public static void fill(boolean[ ] a, boolean val);
     public static void fill(Object[ ] a, Object val);
     public static void fill(float[ ] a, float val);
     public static void fill(long[ ] a, long val);
     public static void fill(int[ ] a, int fromIndex, int toIndex, int val);
     public static void fill(double[ ] a, int fromIndex, int toIndex, 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(float[ ] a, int fromIndex, int toIndex, float val);
     public static void fill(byte[ ] a, int fromIndex, int toIndex, byte val);
     public static void fill(boolean[ ] a, int fromIndex, int toIndex, boolean val);
     public static void fill(Object[ ] a, int fromIndex, int toIndex, Object val);
     public static void fill(long[ ] a, int fromIndex, int toIndex, long val);
5.0  public static int hashCode(short[ ] a);
5.0  public static int hashCode(char[ ] a);
5.0  public static int hashCode(long[ ] a);
5.0  public static int hashCode(int[ ] a);
5.0  public static int hashCode(byte[ ] a);
5.0  public static int hashCode(double[ ] a);
5.0  public static int hashCode(Object[ ] a);
5.0  public static int hashCode(boolean[ ] a);
5.0  public static int hashCode(float[ ] a);
     public static void sort(Object[ ] a);
     public static void sort(short[ ] a);
     public static void sort(float[ ] a);
     public static void sort(double[ ] a);
     public static void sort(long[ ] a);
     public static void sort(byte[ ] a);
     public static void sort(char[ ] a);
     public static void sort(int[ ] a);
     public static <T> void sort(T[ ] a, Comparator<? super T> c);
     public static void sort(short[ ] a, int fromIndex, int toIndex);
     public static void sort(int[ ] a, int fromIndex, int toIndex);
     public static void sort(char[ ] a, int fromIndex, int toIndex);
     public static void sort(long[ ] a, int fromIndex, int toIndex);
     public static void sort(float[ ] a, int fromIndex, int toIndex);
     public static void sort(double[ ] a, int fromIndex, int toIndex);
     public static void sort(byte[ ] a, int fromIndex, int toIndex);
     public static void sort(Object[ ] a, int fromIndex, int toIndex);
     public static <T> void sort(T[ ] a, int fromIndex, int toIndex, Comparator<? super T> c);
5.0  public static String toString(float[ ] a);
5.0  public static String toString(boolean[ ] a);
5.0  public static String toString(Object[ ] a);
5.0  public static String toString(double[ ] a);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
5.0  public static String toString(int[ ] a);
5.0  public static String toString(long[ ] a);
5.0  public static String toString(short[ ] a);
5.0  public static String toString(byte[ ] a);
5.0  public static String toString(char[ ] a);
}
```

BitSet java.util

Java 1.0 ***cloneable serializable***

This class implements an array or list of `boolean` values storing them using a very compact representation that requires only about one 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, and does not even 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 methods described below are only available 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 accomodate 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 zero. 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,

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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 set. 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 non-negative 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) intead. The `intersects( )` method returns `true` if the target `BitSet` and the argument `BitSet` intersect: that is 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` below) 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 preforms 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 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`, hoever). The result is that for all `true` values in the argument `BitSet`, the corresponding values in the target `BitSet` are set to `false`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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: 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 threadsafe.

Figure 16-8. java.util.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
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( );
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Calendar **java.util**

Java 1.1 ***cloneable serializable comparable***

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.

`setTime( )` and the various `set( )` methods set the date represented by a `Calendar` object. 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.

`Calendar` implements `Comparable` in Java 5.0, but not in earlier releases.

Figure 16-9. java.util.Calendar


```
public abstract class Calendar implements Serializable, Cloneable, Comparable<Calendar> {
// 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
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
      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(Locale aLocale);
      public static Calendar getInstance(TimeZone zone);
      public static Calendar getInstance(TimeZone zone, Locale aLocale);
// 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 int getFirstDayOfWeek( );
      public abstract int getGreatestMinimum(int field);
      public abstract int getLeastMaximum(int field);
      public abstract int getMaximum(int field);
      public int getMinimalDaysInFirstWeek( );
      public abstract int getMinimum(int field);
      public final Date getTime( );
      public long getTimeInMillis( );
      public TimeZone getTimeZone( );
      public boolean isLenient( );
      public final boolean isSet(int field);
1.2   public void roll(int field, int amount);
      public abstract void roll(int field, boolean up);
      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 hourOfDay, int minute);
      public final void set(int year, int month, int date, int hourOfDay, int minute,
         int second);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
        public void setFirstDayOfWeek(int value);
        public void setLenient(boolean lenient);
        public void setMinimalDaysInFirstWeek(int value);
        public final void setTime(Date date);
        public void setTimeInMillis(long millis);
        public void setTimeZone(TimeZone value);
    // Methods Implementing Comparable
    5.0  public int compareTo(Calendar anotherCalendar);
    // 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

`java.text.DateFormat.setCalendar( ), javax.xml.datatype.Duration.{addTo( ), getTimeInMillis( ), normalizeWith( )}`

Returned By

`java.text.DateFormat.getCalendar( )`

Type Of

`java.text.DateFormat.calendar`

Collection<E> java.util

Java 1.2 ***collection***

This interface represents a group, or collection, of objects. In Java 5.0 this is a generic interface and the type variable *E* represents the type of the objects in the collection. 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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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`.

Figure 16-10. java.util.Collection<E>


```
public interface Collection<E> extends Iterable<E> {
// Public Instance Methods
     boolean add(E o);
     boolean addAll(Collection<? extends E> c);
     void clear( );
     boolean contains(Object o);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    boolean containsAll(Collection<?> c);
    boolean equals(Object o);
    int hashCode( );
    boolean isEmpty( );
    Iterator<E> iterator( );
    boolean remove(Object o);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    int size( );
    Object[ ] toArray( );
    <T> T[ ] toArray(T[ ] a);
}
```

Implementations

`AbstractCollection`, `List`, `Queue`, `Set`

Passed To

Too many methods to list.

Returned By

Too many methods to list.

Collections java.util

Java 1.2

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 `Comparator` object that reverses the order of another `Comparator` or that reverse the natural ordering of `Comparable` objects.

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.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

The various methods whose names begin with `synchronized` return a threadsafe collection object wrapped around the specified collection. `Vector` and `Hashtable` are the only two collection objects threadsafe 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 Java 5.0, the "checked" methods return wrapped collections that enforce a specified element type for the collection, so that it is not possible to add an element of the wrong type.

In addition to the "synchronized", "unmodifiable", and "checked" 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. In Java 5.0, the `emptySet( )`, `emptyList( )`, and `emptyMap( )` methods are preferred alternatives to these constants, because they are generic methods and return correctly parameterized empty collections. `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 is 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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

of elements in a list, using either an internal source of randomness or the `Random` pseudorandom number generator you provide. In Java 5.0, the `addAll( )` method adds the specified elements to the specified collection. This method is a varargs method and allows elements to be specified in an array or listed individually in the argument list.

Finally, `Collections` defines methods (in addition to the `binarySearch( )` methods described above) that search the elements of a collection: `min( )` and `max( )` methods search an unordered `Collection` for the minimum and maximum elements, according either to 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 the 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. In Java 5.0, `frequency( )` returns the number of occurences of a specified element in a specified collection, and `disjoint( )` determines whether two collections are entirely disjoint—whether they have no elements in common.

```
public class Collections {
// No Constructor
// Public Constants
     public static final List EMPTY_LIST;
1.3  public static final Map EMPTY_MAP;
     public static final Set EMPTY_SET;
// Public Class Methods
5.0  public static <T> boolean addAll(Collection<? super T> c, T ... a);
     public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key);
     public static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c);
5.0  public static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> type);
5.0  public static <E> List<E> checkedList(List<E> list, Class<E> type);
5.0  public static <K,V> Map<K,V> checkedMap(Map<K,V> m, Class<K> keyType, Class<V> valueType);
5.0  public static <E> Set<E> checkedSet(Set<E> s, Class<E> type);
5.0  public static <K,V> SortedMap<K,V> checkedSortedMap(SortedMap<K,V> m, Class<K> keyType,
Class<V> valueType);
5.0  public static <E> SortedSet<E> checkedSortedSet(SortedSet<E> s, Class<E> type);
     public static <T> void copy(List<? super T> dest, List<? extends T> src);
5.0  public static boolean disjoint(Collection<?> c1, Collection<?> c2);
5.0  public static final <T> List<T> emptyList( );
5.0  public static final <K,V> Map<K,V> emptyMap( );
5.0  public static final <T> Set<T> emptySet( );
     public static <T> Enumeration<T> enumeration(Collection<T> c);
     public static <T> void fill(List<? super T> list, T obj);
5.0  public static int frequency(Collection<?> c, Object o);
1.4  public static int indexOfSubList(List<?> source, List<?> target);
1.4  public static int lastIndexOfSubList(List<?> source, List<?> target);
1.4  public static <T> ArrayList<T> list(Enumeration<T> e);
     public static <T extends Object&Comparable<? super T>> T max(Collection<? extends T> coll);
     public static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp);
     public static <T extends Object&Comparable<? super T>> T min(Collection<? extends T> coll);
     public static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp);
     public static <T> List<T> nCopies(int n, T o);
1.4  public static <T> boolean replaceAll(List<T> list, T oldVal, T newVal);
     public static void reverse(List<?> list);
     public static <T> Comparator<T> reverseOrder( );
5.0  public static <T> Comparator<T> reverseOrder(Comparator<T> cmp);
1.4  public static void rotate(List<?> list, int distance);
     public static void shuffle(List<?> list);
     public static void shuffle(List<?> list, Random rnd);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
     public static <T> Set<T> singleton(T o);
1.3  public static <T> List<T> singletonList(T o);
1.3  public static <K,V> Map<K,V> singletonMap(K key, V value);
     public static <T extends Comparable<? super T>> void sort(List<T> list);
     public static <T> void sort(List<T> list, Comparator<? super T> c);
1.4  public static void swap(List<?> list, int i, int j);
     public static <T> Collection<T> synchronizedCollection(Collection<T> c);
     public static <T> List<T> synchronizedList(List<T> list);
     public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
     public static <T> Set<T> synchronizedSet(Set<T> s);
     public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);
     public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);
     public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c);
     public static <T> List<T> unmodifiableList(List<? extends T> list);
     public static <K,V> Map<K,V> unmodifiableMap(Map<? extends K,? extends V> m);
     public static <T> Set<T> unmodifiableSet(Set<? extends T> s);
     public static <K,V> SortedMap<K,V> unmodifiableSortedMap(SortedMap<K,? extends V> m);
     public static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> s);
}
```

Comparator<T> java.util

Java 1.2

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. `Comparator` has been made generic in Java 5.0 and the type variable *`T`* represents the type of objects being compared.

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`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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<T> {
// Public Instance Methods
     int compare(T o1, T o2);
     boolean equals(Object obj);
}
```

Implementations

`java.text.Collator`

Passed To

`Arrays.{binarySearch( ),sort( )},Collections.{binarySearch( ), max( ),min( ),reverseOrder( ),sort( )}, PriorityQueue.PriorityQueue( ),TreeMap.TreeMap( ), TreeSet.TreeSet( ), java.util.concurrent.PriorityBlockingQueue.PriorityBlockingQueue ( )`

Returned By

`Collections.reverseOrder( ),PriorityQueue.comparator( ), SortedMap.comparator( ),SortedSet.comparator( ), TreeMap.comparator( ),TreeSet.comparator( ), java.util.concurrent.PriorityBlockingQueue.comparator( )`

Type Of

`String.CASE_INSENSITIVE_ORDER`

ConcurrentModificationException java.util

Java 1.2 ***serializable unchecked***

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.

Figure 16-11. java.util.ConcurrentModificationException


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


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public ConcurrentModificationException(String message);
}
```

Currency **java.util**

Java 1.4 ***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`. The symbol for a USD would be "$" in a U.S locale, but might be "US$" in other locales, for example. 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 which are traditionally divided into thousandths, and 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 website of the "maintenance agency" for this standard: http://www.iso.org/iso/en/prods-services/popstds/currencycodeslist.html.

Figure 16-12. java.util.Currency


```
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( );
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Passed To

`java.text.DecimalFormat.setCurrency( ),`
`java.text.DecimalFormatSymbols.setCurrency( ),`
`java.text.NumberFormat.setCurrency( )`

Returned By

`java.text.DecimalFormat.getCurrency( ),`
`java.text.DecimalFormatSymbols.getCurrency( ),`
`java.text.NumberFormat.getCurrency( )`

Date java.util

Java 1.0 ***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.

Figure 16-13. java.util.Date


```
public class Date implements Serializable, Cloneable, Comparable<Date> {
// Public Constructors
     public Date( );
     public Date(long date);
#    public Date(String s);
#    public Date(int year, int month, int date);
#    public Date(int year, int month, int date, int hrs, int min);
#    public Date(int year, int month, int date, int hrs, int min, int sec);
// Public Instance Methods
     public boolean after(Date when);
     public boolean before(Date when);
     public long getTime( );                              default:1101702237486
     public void setTime(long time);
// Methods Implementing Comparable
1.2  public int compareTo(Date anotherDate);
// Public Methods Overriding Object
1.2  public Object clone( );
     public boolean equals(Object obj);
     public int hashCode( );
     public String toString( );
// Deprecated Public Methods
#    public int getDate( );                             default:28
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
#    public int getDay( );                              default:0
#    public int getHours( );                            default:20
#    public int getMinutes( );                                      default:23
#    public int getMonth( );                            default:10
#    public int getSeconds( );                                      default:57
#    public int getTimezoneOffset( );                               default:480
#    public int getYear( );                             default:104
#    public static long parse(String s);
#    public void setDate(int date);
#    public void setHours(int hours);
#    public void setMinutes(int minutes);
#    public void setMonth(int month);
#    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);
}
```

Passed To

Too many methods to list.

Returned By

Too many methods to list.

Dictionary<K,V> java.util

Java 1.0

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

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

Subclasses

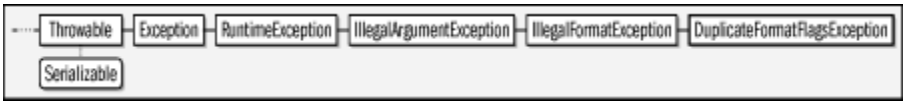
`Hashtable`

DuplicateFormatFlagsException java.util

Java 5.0 ***serializable unchecked***

An `IllegalFormatException` of this type is thrown by a `Formatter` when the format string contains duplicate format flags for the same conversion specifier.

Figure 16-14. java.util.DuplicateFormatFlagsException



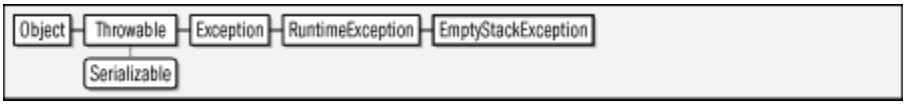

```
public class DuplicateFormatFlagsException extends IllegalFormatException {
// Public Constructors
     public DuplicateFormatFlagsException(String f);
// Public Instance Methods
     public String getFlags( );
// Public Methods Overriding Throwable
     public String getMessage( );
}
```

EmptyStackException java.util

Java 1.0 ***serializable unchecked***

Signals that a `Stack` object is empty.

Figure 16-15. java.util.EmptyStackException




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

Enumeration<E> java.util

Java 1.0

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. This interface is superseded in Java 1.2 by the `Iterator` inteface. In Java 5.0 this interface has been made generic and defines the type variable *`E`* to represent the type of the objects being enumerated.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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.

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<E> {
// Public Instance Methods
     boolean hasMoreElements( );
     E nextElement( );
}
```

Implementations

`StringTokenizer`

Passed To

`java.io.SequenceInputStream.SequenceInputStream( )`, `Collections.list( )`

Returned By

Too many methods to list.

EnumMap<K extends Enum<K>,V> java.util

Java 5.0 ***cloneable serializable collection***

This class is a `Map` implementation for use with enumerated types. The key type *`K`* must be an enumerated type, and all keys must be enumerated constants defined by that type. `null` keys are not permitted. The value type *`V`* is unrestricted and `null` values are permitted.

The `EnumMap` implementation is based on an array of elements of type *`V`*. The length of this array is the same as the number of constants defined by the enumerated type *`K`*. All `Map` operations execute in constant time. The iterators of the `keySet( )`, `entrySet( )`, and `values( )` collections iterate their elements in the ordinal order of the enumerated constants. `EnumMap` is not threadsafe, but its iterators are based on a snapshot of the underlying array and never throw `ConcurrentModificationException`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-16. java.util.EnumMap<K extends Enum<K>,V>


```
public class EnumMap<K extends Enum<K>,V>
        extends AbstractMap<K,V> implements Serializable, Cloneable {
// Public Constructors
     public EnumMap(EnumMap<K,? extends V> m);
     public EnumMap(Class<K> keyType);
     public EnumMap(Map<K,? extends V> m);
// Public Instance Methods
     public EnumMap<K,V> clone( );
     public V put(K key, V value);
// Public Methods Overriding AbstractMap
     public void clear( );
     public boolean containsKey(Object key);
     public boolean containsValue(Object value);
     public Set<Map.Entry<K,V>> entrySet( );
     public boolean equals(Object o);
     public V get(Object key);
     public Set<K> keySet( );
     public void putAll(Map<? extends K,? extends V> m);
     public V remove(Object key);
     public int size( );
     public Collection<V> values( );
}
```

EnumSet<E extends Enum<E>> **java.util**

Java 5.0 ***cloneable serializable collection***

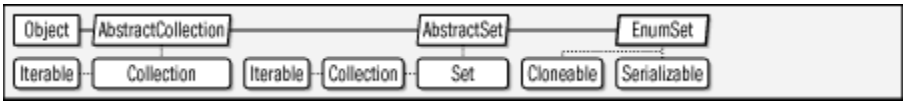
This `Set` implementation is specialized for use with enumerated constants. The element type *E* must be an enumerated type, and `null` is not allowed as a member of the set.

`EnumSet` does not define a constructor. Instead, it defines various static factory methods for creating sets. Use one of the `of( )` methods for creating an `EnumSet` and initializing its elements. For efficiency, versions of this method that accept one through five arguments are defined. If you pass more than five arguments, the varargs version will be invoked. The `allOf( )` and `noneOf( )` methods define full and empty sets but require the `Class` of the enumerated type since they do not have any other arguments to define the element type. `complementOf( )` returns an `EnumSet` that contains all enumerated constants not contained by the specified `EnumSet`. The `range( )` factory creates a set that includes the two specified values and any enumerated constants that fall between them in the enumerated type declaration. (Note that this definition of a range includes both endpoints and differs from most Java methods, in which the second argument specifies the first value past the end of the range.)

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

The `EnumSet` implementation is based on a bit vector that includes one bit for each constant defined by the enumerated type *`E`*. Because of this compact and efficient representation, basic `Set` operations occur in constant time, and the `Iterator` returns enumerated constants in the order in which they are declared in the type *`E`*. `EnumSet` is not threadsafe, but the `Iterator` uses a copy of the internal bit vector and never throws `ConcurrentModificationException`.

Figure 16-17. java.util.EnumSet<E extends Enum<E>>




```
public abstract class EnumSet<E extends Enum<E>>
        extends AbstractSet<E> implements Cloneable, Serializable {
// No Constructor
// Public Class Methods
     public static <E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType);
     public static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> s);
     public static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> s);
     public static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c);
     public static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType);
     public static <E extends Enum<E>> EnumSet<E> of(E e);
     public static <E extends Enum<E>> EnumSet<E> of(E first, E ... rest);
     public static <E extends Enum<E>> EnumSet<E> of(E e1, E e2);
     public static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3);
     public static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4);
     public static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5);
     public static <E extends Enum<E>> EnumSet<E> range(E from, E to);
// Public Instance Methods
     public EnumSet<E> clone( );
}
```

EventListener java.util

Java 1.1 ***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 {
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Implementations

`EventListenerProxy`, `java.util.prefs.NodeChangeListener`, `java.util.prefs.PreferenceChangeListener`, `javax.net.ssl.HandshakeCompletedListener`, `javax.net.ssl.SSLSessionBindingListener`

Passed To

`EventListenerProxy.EventListenerProxy( )`

Returned By

`EventListenerProxy.getListener( )`

EventListenerProxy java.util

Java 1.4

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.

Figure 16-18. java.util.EventListenerProxy


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

EventObject java.util

Java 1.1 ***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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

some way, generated the event. The source object is passed to the `EventObject( )` constructor and is returned by the `getSource( )` method.

Figure 16-19. java.util.EventObject


```
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

`java.util.prefs.NodeChangeEvent`,
`java.util.prefs.PreferenceChangeEvent`,
`javax.net.ssl.HandshakeCompletedEvent`,
`javax.net.ssl.SSLSessionBindingEvent`

FormatFlagsConversionMismatchException java.util

Java 5.0 ***serializable unchecked***

An `IllegalFormatException` of this type is thrown by a `Formatter` when a conversion specifier and a format flag specified with it are incompatible.

Figure 16-20. java.util.FormatFlagsConversionMismatchException


```
public class FormatFlagsConversionMismatchException extends IllegalFormatException {
// Public Constructors
     public FormatFlagsConversionMismatchException(String f, char c);
// Public Instance Methods
     public char getConversion( );
     public String getFlags( );
// Public Methods Overriding Throwable
     public String getMessage( );
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Formattable **java.util**

Java 5.0

This interface should be implemented by classes that want to interact with the `Formatter` class more intimately than is possible with the `toString` method. When a `Formattable` object is the argument for a `%s` or `%S` conversion, its `formatTo( )` method is invoked rather than its `toString( )` method. `formatTo( )` is responsible for formatting a textual representation of the object to the specified *`formatter`*, subject to the constraints imposed by the *`flags`*, *`width`*, and *`precision`* arguments.

The *`flags`* argument is a bitmask of zero or more `FormattableFlags` constants. Each flag provides information about the format specification that resulted in the invocation of `formatTo( ).FormattableFlags.ALTERNATE` indicates that the # flag was used and that the `Formattable` should format itself using some alternate form. The interpretation of the alternate form is entirely up to the `Formattable` implementation. `LEFT_JUSTIFY` means that the – flag was used and that the `Formattable` should pad its output on the right, instead of on the left. `UPPERCASE` indicates that the `%S` conversion was used instead of `%s` and the `Formattable` should output uppercase characters instead of lowercase.

The *`width`* and *`precision`* arguments specify the width and precision specified along with the `%s` format specifier, or -1 if no width and precision are specified. The `Formattable` object should treat these values the same way that `Formatter` does. The text to be output should first be truncated to fit within *`precision`* characters and then padded on the left (or right if the `LEFT_JUSTIFY` flag is set) with spaces for a total length of *`width`* characters. Note that a `Formattable` implementation may fulfill the obligations imposed by the `LEFT_JUSTIFY` and `UPPERCASE` flags and the *`width`* and *`precision`* arguments by constructing a suitable format string to pass back to the specified `Formatter`.

If a `Formattable` implementation wants to perform locale-specific formatting, it can query the `Locale` of the `Formatter` with the `locale( )` method. Note, however, that the returned value is the locale specified when the `Formatter` was created, not the `Locale`, if any, passed to the `format( )` method. There is no way for a `Formattable` object to access that `Locale`.

```
public interface Formattable {
// Public Instance Methods
     void formatTo(java.util.Formatter formatter, int flags, int width, int precision);
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

FormattableFlags java.util

Java 5.0

This class defines three constants representing flags that may be passed as a bitmask to the `Formattable.formatTo( )` method. See `Formattable` for the interpretation of these flags.

```
public class FormattableFlags {
// No Constructor
// Public Constants
     public static final int ALTERNATE;                                =4
     public static final int LEFT_JUSTIFY;                             =1
     public static final int UPPERCASE;                                =2
}
```

Formatter java.util

Java 5.0 ***closeable flushable***

The `Formatter` class is a utility for formatting text in the style of the `printf( )` method of the C programming language. Every `Formatter` has an associated `java.lang.Appendable` object (such as a `StringBuilder` or `PrintWriter`) that is specified when the `Formatter` is created. `format( )` is a varargs method that expects a "format string" argument followed by some number of `Object` arguments. The format string uses a grammar, described in detail later in the entry, to specify how the arguments that follow are to be converted to strings. After the arguments are converted, they are substituted into the format string, and the resulting text is appended to the `Appendable`. A variant of the `format( )` method accepts a `Locale` object that can affect the argument conversions.

For ease of use, a `Formatter` never throws a `java.io.IOException`, even when the underlying `Appendable` throws one. When using a `Formatter` with a stream-based `Appendable` object that may throw an `IOException`, you can use the `ioException( )` method to obtain the most recently thrown exception, or `null` if no exception has been thrown by the `Appendable`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`Formatter` implements the `Closeable` and `Flushable` interfaces of the `java.io` package, and its `close( )` and `flush( )` methods call the corresponding methods on its `Appendable` object, if that object itself implements `Closeable` or `Flushable`. When a `Formatter` sends its output to a stream or similar `Appendable`, remember to call `close( )` when you are done with it. It is always safe to call `close( )` even if the underlying `Appendable` is not `Closeable`. Note that once a `Formatter` has been closed, no other method except `ioException( )` may be called.

`locale( )` returns the `Locale` passed to the `Formatter( )` constructor or `null`. `out( )` returns the `Appendable` that this `Formatter` sends its output to. `toString( )` returns the result of calling `toString( )` on that `Appendable`. This is useful when the `Appendable` is a `StringBuilder`, for example, as it is when the no-argument version of the `Formatter( )` constructor is used. If the `Appendable` is a stream class, however, the `toString( )` method is not typically useful.

Note that the Java 5.0 API provides a number of convenience methods that use the `Formatter` class, and in many cases it is unnecessary to create a `Formatter` object explicitly. See the static `String.format( )` method and the `format( )` and `printf( )` methods of `java.io.PrintWriter` and `java.io.PrintStream`.

If you do need to create a `Formatter` object explicitly, you can choose from a number of constructors. The most general case is to pass the desired `Appendable` or the desired `Locale` and `Appendable` objects to the constructor. The no-argument constructor is a convenience that creates a `StringBuilder` to append to. Obtain this `StringBuilder` with `out( )` or obtain its contents as a `String` with `toString( )`. If you specify a single `Locale` argument, the resulting `Formatter` uses the specified locale with a `StringBuilder`.

You can use a `Formatter` to write formatted output to a file by specifying either the `File` object or filename as a `String`. Variants of these constructors allow you to specify the name of the charset to use for character-to-byte conversion and also a `Locale`. Note that these methods overwrite existing files rather than appending to them. Other constructors create an `Appendable` object for you based on the `java.io.OutputStream` or `java.io.PrintStream` you specify. In the `OutputStream` case, you may optionally specify the charset to use or the charset and a `Locale`.

The Format String and Format Specifiers

The API for `Formatter` and `Formatter`-based convenience methods is relatively simple. The power of these formatting methods lies in the format string that is the first argument

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

(or second argument if a `Locale` is specified) to the various `format( )` and `printf( )` methods. The format string may contain any amount of regular text, which is printed or appended literally to the destination `Appendable` object. This plain text may be interspersed with *format specifiers* which specify how a subsequent argument is to be formatted as a string. In contrast to the simple API, the grammar for these format specifiers is surprisingly complex. Experienced C programmers will find that the grammar is largely compatible with the `printf( )` format string grammar of the standard C library.

Each format specifier begins with a percent sign and ends with a one- or two-character conversion type that specifies most of the details of the conversion and formatting. In between these two are optional flags that provide additional details about how the formatting should be done. The general syntax of a format specifier is as follows. Square brackets indicate optional items:

```
%[argument][flags][width][.precision]type
```

Note that the percent sign and the *`type`* are the only two required portions of a format specifier. We begin, therefore, with a listing of conversion types (see Table 16-1). A discussion of *`argument`*, *`flags`*, *`width`*, and *`precision`* follows. In the table of conversion types below, if uppercase and lowercase variants of the type specifier are listed together, the uppercase variant produces the same output as the lowercase variant except that all lowercase letters are converted to uppercase. Note that `format( )` never throws `NullPointerException` because of `null` arguments following the format string. A `null` argument is formatted as "null" or "NULL" for all conversion characters except `%b` and `%B`, which produce "false" or "FALSE".

Table 16-1. Formatter conversion types

Conversion	Description
Simple conversions	
`%%`	Outputs a single percent sign.This is simply an escape sequence used to embed percent signs literally in the output string. This conversion does not use an argument.
`%n`	Outputs the platform-specific line separator. This conversion represents the value returned by `System.getProperty("line.separator")`. This conversion does not use an argument.
`%s, %S`	Formats and outputs the argument as a string, optionally converting it to uppercase for the `%S` conversion. The argument may be of any type. If the argument implements `Formattable`, its `formatTo( )` method is called to perform the formatting. Otherwise, its `toString( )` method is called to convert it to a string. If the argument is `null`, the output string is "null" or "NULL".
`%c, %C`	Outputs the argument as a single character. The argument type must be `Byte`, `Short`, `Character`, or `Integer`. The argument value must represent a valid Unicode code point. (See `Character.isValidCodePoint( )`.)
`%b, %B`	Outputs the argument value as the string "true" or "false" (or "TRUE" or "FALSE"). The argument may be of any type and any value. If it is a `Boolean` argument, the output reflects the argument value. Otherwise, if the argument is `null`, the output is "false" or "FALSE". For any other value, the output is "true" or "TRUE". Note that this differs from normal Java conversions in which `boolean` values are not convertible to or from any other type.
`%h, %H`	Outputs the hexadecimal representation of the hashcode for the argument. Arguments of any type and value are allowed. This conversion type is useful mainly for debugging.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Conversion	Description
Numeric Conversions	
`%d`	Formats the argument as a base-10 integer. The argument must be a `Byte`, `Short`, `Integer`, `Long`, or `BigInteger`.
`%o`	Formats the argument as a base-8 octal integer. The allowed argument types are the same as for `%d`. For any argument type other than `BigInteger`, the value is treated as unsigned.
`%x, %X`	Formats the argument as a base-16 hexadecimal integer. The allowed argument types and values are the same as for `%d`. For any argument type other than `BigInteger`, the value is treated as unsigned.
`%e, %E`	Formats the argument as a base-10 floating-point number, using exponential notation. The output consists of a single digit, a locale-specific decimal point, and the number of fractional digits specified by the *`precision`* of the format specifier, or six fractional digits if no *`precision`* is specified. These digits are followed by the letter `e` or `E` and the exponent of the number. The argument must be a `Float`, `Double`, or `BigDecimal`. The values `NaN` and `Infinity` are formatted as "NaN" and "Infinity" or their uppercase equivalents.
`%f`	Formats the argument as a floating-point number in base-10, without using exponential notation. If the number is large, this may produce quite a few digits. Because exponential notation is never used, the output will never include a letter, and there is no uppercase variant of this conversion. Legal argument types and special-case values are as for `%e`.
`%g, %G`	Formats the argument as a base-10 floating-point number, displaying no more than the number of significant digits specified by the *`precision`* of the format specifier, or no more than 6 significant digits if no *`precision`* is specified. If the value has more than the allowed number of significant digits, it is printed using exponential notation (see `%e`) to limit the display to the specified number of digits. Otherwise, all digits of the value are printed explicitly as they would be with the `%f` conversion type. Legal argument types and special case values are as for `%e`.
`%a, %A`	Formats the argument in hexadecimal floating-point format. Legal argument types and special case values are as for `%e`.
Dates and Times	
`%t, %T`	All date and time format types are two-letter codes beginning with `%t` or `%T`. The specific format types are listed below, in alphabetical order, using `%t` as the prefix. For uppercase, use `%T` instead. Upper- and lowercase variants of the second letter of a time or date format type are sometimes completely unrelated. Other times, the lowercase conversion produces an abbreviation of the value produced by the uppercase conversion. The argument for a date or time conversion must be a `Date`, `Calendar`, or `Long`. In the case of `Long`, the value is interpreted as milliseconds since the epoch, as in `System currentTimeMillis( )`.
`%tA`	The locale-specific full name of the day of the week.
`%ta`	The locale-specific abbreviation of the day of the week.
`%tB`	The locale-specific name of the month. See `%tm`.
`%tb`	The locale-specific abbreviation for the month.
`%tC`	The century: the year divided by 100, with leading zeros if necessary to produce a value from 00 to 99
`%tc`	The complete date and time. Equivalent to "%ta %tb %td %tT %tZ %tY".
`%tD`	The date in a short numeric form used in the US locale. Equivalent to "%tm/%td/%ty".
`%td`	The day of the month, as a two-digit number between 01 and 31. See `%te`.
`%tE`	The date expressed as milliseconds since Midnight UTC on January 1st, 1970.
`%te`	The day of the month as a one- or two-digit number without leading zeros between 1 and 31. See `%td`.
`%tF`	The numeric date in ISO8601 format: %tY-%tm-%td.
`%tH`	Hour of the day using a 24-hour clock, formatted as two digits between 00 and 23. See `%tI`.
`%th`	The abbreviated month name. Same as `%tb`.
`%tI`	Hour of the day using a 12-hour clock, formatted as two digits between 01 and 12. See `%tH` and `%tP`.
`%tj`	The day of the year as three digits with leading zeros if necessary: 001-366
`%tk`	Hour of the day on a 24-hour clock using one or two digits without a leading zero: 0-23. See `%tl`.
`%tL`	Milliseconds within the second, expressed as three digits with leading zeros: 000-999.
`%tl`	Hour of the day on a 12-hour clock using one or two digits without a leading zero: 1-12.
`%tM`	Minute within the hour as two digits with a leading zero if necessary: 00-59.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Conversion	Description
%tm	The month of the year as a two-digit number between 01 and 12, or between 01 and 13 for lunar calendars. See %tB and %tb.
%tN	Nanosecond within the second, expressed as nine digits with leading zeros if necessary. Note that platforms are not required to able to resolve times with nanosecond precision.
%tP	The locale-specific morning or afternoon indicator (such as "am" or "pm") used with 12-hour clocks. %tP uses lowercase and %TP uses uppercase.
%tp	Like %tP but uses uppercase for both %tp and %Tp variants.
%tR	The hour and minute on a 24-hour clock. Equivalent to "%tH:%tM".
%tr	The hour, minute, and second on a 12-hour clock. Equivalent to "%tI:%tM:%tS %tP" except that the am/pm indicator %tP may be in a different locale-dependent position.
%tS	Seconds within the minute, as two digits with a leading zero if necessary. The range is normally 00-59, but a value of 60 is allowed for leap seconds.
%ts	Seconds since the beginning of the epoch. See %tE.
%tT	The time in hours, minutes, and seconds using 24-hour format. Equivalent to "%tH:%tM:%tS".
%tY	The year, using at least four digits, formatted with leading zeros, if necessary.
%ty	The last two digits of the year, 00-99
%tZ	An abbreviation for the time zone.
%tz	The time zone as numeric offset from GMT.

Argument Specifier

Every format specifier in a format string except for %% and %n requires an argument that contains the value to format. These arguments follow the format string in the call to format( ) or printf( ). By default, a format specifier uses the next unused argument. In the following printf( ) call, the first and second %s format specifiers format the second and third arguments, respectively:

```
out.printf("Name: %s %s%n", first, last);
```

If a format specifier includes the character < after the %, it specifies that the argument of the previous format specifier should be reused. This allows the same object (such as a date) to be formatted more than once (yielding a formatted date and time, for example):

```
out.printf("Date: %tD%nTime: %<tr%n", System.currentTimeMillis( ));
```

It is an error to use < in the first format specifier of a format string.

Argument numbers may also be specified absolutely. If the % sign is followed by one or more digits and a $ sign, those digits specify an argument number. For example %1$d specifies that the first argument following the format string should be formatted as an integer. Absolute argument numbers are particularly useful for localization since the different translations of a message may need to interpolate the arguments in a different order. The following example includes a format string that might be used in a locale where a person's family name is typically printed (in uppercase) before the given name. Note that the arguments are not passed in the same order that they are formatted.

```
String name = String.format("%2$S, %1$s", firstname, lastname);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Neither absolute argument indexing with a number and $ character or relative argument indexing with < affect the order in which arguments are interpolated for format specifiers that use neither $ or <. The first format specifier that has neither an absolute or relative argument specification uses the first argument following the format string, regardless of what has come before. The code above could be rewritten like this, for example:

```
String name = String.format("%2$S, %s", firstname, lastname);
```

Flags

Following the optional argument specifier, a format specifier may include one or more flag characters. The defined flags, their effects, and the format types for which they are legal are specified in Table 16-2:

Table 16-2. Formatter flags

Flag	Description
-	A hyphen specifies that the formatted value should be left-justified within the specified *width*. This flag can be used with any conversion type except %n as long as the conversion specifier also includes a *width* (see below). When a width is specified without this flag, the formatted string is padded on the left to produce right-justified output.
#	The # flag specifies that output should appear in an "alternate form" that depends on the type being formatted. For %o conversions, this flag specifies that the output should include a leading 0. For %x and %X conversions, it specifies that output should include a leading 0x or 0X. For the %s and %S conversions, the # flag may be used if the argument implements Formattable. In this case, the flag is passed on to the formatTo( ) method of the argument, and it is up to that formatTo( ) method to produce its output in some alternate form.
+	This flag specifies that numeric output should always include a sign: a value that is nonnegative will have "+" added in front of it. This flag may be used with any numeric conversion that may yield a signed result. This includes %d, %e, %f, %g, %a, and their uppercase variants. It also includes %o, %x, and %X conversions applied to BigInteger arguments.
	The space character is a (hard-to-read) flag that specifies that non-negative values should be prefixed with a space. This flag may be used with the same conversion and argument types as the + flag, and is useful when aligning positive and negative numbers in a column
(	This flag specifies that negative numbers should be enclosed in parentheses, as is commonly done in financial statements, for example. This flag may be used with the same format and argument types as the + flag, except that it may not be used with %a conversions.
0	The digit zero, used as a flag, specifies that numeric values should be padded on the left (after the sign character, if any) with zeros. This flag may be used only if a width is specified, and may not be used in conjunction with the - flag.
,	This flag specifies that numbers should be formatted using the locale-specific grouping separator. In the US locale, for example, a comma would appear every three digits to separate the number into thousands, millions, and so on. This flag may be used with %d, %e, %E, %f, %g, and %G conversions only.

Width

The *width* portion of a format specifier is one or more digits that specify the minimum number of characters to be produced. If the formatted value is narrower than the specified width, (by default) it is padded on the left with spaces, producing a right-justified value. The - and 0 flags can be used to specify left-justification or padding with zeros instead.

A width may be specified with any format type except %n.

Precision

The *precision* portion of a format specifier is one or more digits following a decimal point. The meaning of this number depends on which format type it is used with:

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

- For `%e`, `%E`, and `%f`, the precision specifies the number of digits to appear after the decimal point. Zeros are appended on the right, if necessary. The default precision is 6.
- For `%g` and `%G` format types, the precision specifies the total number of significant digits to be displayed. As a corollary, it specifies the largest and smallest values that can be displayed without resorting to exponential notation. The default precision is 6. If a precision of 0 is specified, it is treated as a precision of 1.
- For `%s`, `%h` and `%b` format types, and their uppercase variants, the precision specifies the maximum number of characters to be output. If no precision is specified, there is no maximum. If the formatted output would exceed the *`precision`* of characters, it is truncated. If *`precision`* is smaller than *`width`*, the formatted value is first truncated as necessary and then padded within the specified *`width`*.
- Specifying a precision for any other conversion type causes an exception at runtime.

Synopsis

Figure 16-21. java.util.Formatter


```
public final class Formatter implements java.io.Closeable, java.io.Flushable {
// Public Constructors
     public Formatter( );
     public Formatter(java.io.PrintStream ps);
     public Formatter(java.io.OutputStream os);
     public Formatter(java.io.File file) throws java.io.FileNotFoundException;
     public Formatter(String fileName) throws java.io.FileNotFoundException;
     public Formatter(Locale l);
     public Formatter(Appendable a);
     public Formatter(java.io.OutputStream os, String csn)
        throws java.io.UnsupportedEncodingException;
     public Formatter(java.io.File file, String csn)
        throws java.io.FileNotFoundException, java.io.UnsupportedEncodingException;
     public Formatter(Appendable a, Locale l);
     public Formatter(String fileName, String csn)
        throws java.io.FileNotFoundException, java.io.UnsupportedEncodingException;
     public Formatter(String fileName, String csn, Locale l)
        throws java.io.FileNotFoundException, java.io.UnsupportedEncodingException;
     public Formatter(java.io.File file, String csn, Locale l)
        throws java.io.FileNotFoundException, java.io.UnsupportedEncodingException;
     public Formatter(java.io.OutputStream os, String csn, Locale l)
        throws java.io.UnsupportedEncodingException;
// Nested Types
     public enum BigDecimalLayoutForm;
// Public Instance Methods
     public java.util.Formatter format(String format, Object... args);
     public java.util.Formatter format(Locale l, String format, Object... args);
     public java.io.IOException ioException( );
     public Locale locale( );
     public Appendable out( );
// Methods Implementing Closeable
     public void close( );
// Methods Implementing Flushable
     public void flush( );
// Public Methods Overriding Object
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public String toString( );
}
```

Passed To

`Formattable.formatTo( )`

Formatter.BigDecimalLayoutForm java.util

Java 5.0 ***serializable comparable enum***

This enumerated type is intended for internal use by the `Formatter` class, but was inadvertently declared `public`. This type serves no useful purpose and should not be used. It will likely be removed in a future release.

```
public enum Formatter.BigDecimalLayoutForm {
// Enumerated Constants
    SCIENTIFIC,
    DECIMAL_FLOAT;
// Public Class Methods
    public static Formatter.BigDecimalLayoutForm valueOf(String name);
    public static final Formatter.BigDecimalLayoutForm[ ] values( );
}
```

FormatterClosedException java.util

Java 5.0 ***serializable unchecked***

An exception of this type is thrown when an attempt is made to use a `Formatter` whose `close( )` method has been called.

Figure 16-22. java.util.FormatterClosedException


```
public class FormatterClosedException extends IllegalStateException {
// Public Constructors
    public FormatterClosedException( );
}
```

GregorianCalendar java.util

Java 1.1 ***cloneable serializable comparable***

This concrete subclass of `Calendar` implements the standard solar calendar with years numbered from the birth of Christ that is used is 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.

Figure 16-23. java.util.GregorianCalendar


```
public class GregorianCalendar extends Calendar {
// Public Constructors
     public GregorianCalendar( );
     public GregorianCalendar(Locale aLocale);
     public GregorianCalendar(TimeZone zone);
     public GregorianCalendar(TimeZone zone, Locale aLocale);
     public GregorianCalendar(int year, int month, int dayOfMonth);
     public GregorianCalendar(int year, int month, int dayOfMonth, int hourOfDay,
        int minute);
     public GregorianCalendar(int year, int month, int dayOfMonth, int hourOfDay, int minute,
        int second);
// Public Constants
     public static final int AD;                                          =1
     public static final int BC;                                          =0
// Public Instance Methods
     public final Date getGregorianChange( );
     public boolean isLeapYear(int year);
     public void setGregorianChange(Date date);
// Public Methods Overriding Calendar
     public void add(int field, int amount);
5.0  public Object clone( );
     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);
5.0  public TimeZone getTimeZone( );
     public int hashCode( );
     public void roll(int field, boolean up);
1.2  public void roll(int field, int amount);
5.0  public void setTimeZone(TimeZone zone);
// Protected Methods Overriding Calendar
     protected void computeFields( );
     protected void computeTime( );
}
```

Passed To

`javax.xml.datatype.DatatypeFactory.newXMLGregorianCalendar( )`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Returned By

`javax.xml.datatype.XMLGregorianCalendar.toGregorianCalendar( )`

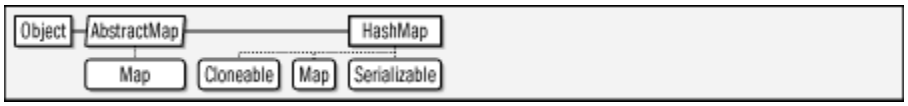
HashMap<K,V> java.util

Java 1.2 ***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 multithreaded 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`.

Figure 16-24. java.util.HashMap<K,V>




```
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable {
// Public Constructors
     public HashMap( );
     public HashMap(int initialCapacity);
     public HashMap(Map<? extends K,? extends V> 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<Map.Entry<K,V>> entrySet( );
     public V get(Object key);
     public boolean isEmpty( );                          default:true
     public Set<K> keySet( );
     public V put(K key, V value);
     public void putAll(Map<? extends K,? extends V> m);
     public V remove(Object key);
     public int size( );
     public Collection<V> values( );
// Public Methods Overriding AbstractMap
     public Object clone( );
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Subclasses

`LinkedHashMap`

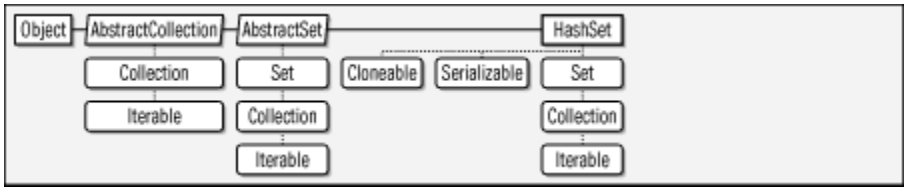
HashSet<E> java.util

Java 1.2 ***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 the default value. See `Set` and `Collection` for details on the methods of `HashSet`. See also `TreeSet` and `HashMap`.

Figure 16-25. java.util.HashSet<E>




```
public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable {
// Public Constructors
     public HashSet( );
     public HashSet(Collection<? extends E> c);
     public HashSet(int initialCapacity);
     public HashSet(int initialCapacity, float loadFactor);
// Methods Implementing Set
     public boolean add(E o);
     public void clear( );
     public boolean contains(Object o);
     public boolean isEmpty( );                          default:true
     public Iterator<E> iterator( );
     public boolean remove(Object o);
     public int size( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
// Public Methods Overriding Object
     public Object clone( );
}
```

Subclasses

`LinkedHashSet`

Hashtable<K,V> java.util

Java 1.0 ***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. In Java 1.2 and later `Hashtable` has been modified to impement the `Map` interface. The `HashMap` class is typically preferred over this one, although the `synchronized` methods of this class are useful in multi-threaded applications. (But see `java.util.concurrent.ConcurrentHashMap`.) In Java 5.0 this class has been made generic along with the `Map` interface. The type variable *`K`* represents the type of the hashtable keys and the type variable *`V`* represents the type of the hashtable values.

`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`.

Figure 16-26. java.util.Hashtable<K,V>


```
public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>,
        Cloneable, Serializable {
// Public Constructors
     public Hashtable( );
1.2  public Hashtable(Map<? extends K,? extends V> t);
     public Hashtable(int initialCapacity);
     public Hashtable(int initialCapacity, float loadFactor);
// Public Instance Methods
     public void clear( );          Implements:Map synchronized
     public boolean contains(Object value);          synchronized
     public boolean containsKey(Object key);         Implements:Map synchronized
     public V get(Object key);    Implements:Map synchronized
     public boolean isEmpty( );     Implements:Map synchronized default:true
     public V put(K key, V value);                   Implements:Map synchronized
     public V remove(Object key);                    Implements:Map synchronized
     public int size( );            Implements:Map synchronized
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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

Subclasses

`Properties`

IdentityHashMap<K,V> java.util

Java 1.4 ***cloneable serializable collection***

This `Map` implementation has a API that is very similar to `HashMap`, and uses an internal hashtable, like `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—whether they are exactly the same object. This one difference in how key equality is tested has profound 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 what is required.

Figure 16-27. java.util.IdentityHashMap<K,V>


```
public class IdentityHashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>,
        Serializable, Cloneable {
```

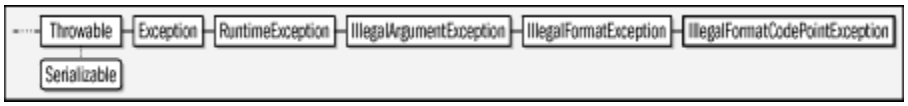
```
// Public Constructors
     public IdentityHashMap( );
     public IdentityHashMap(int expectedMaxSize);
     public IdentityHashMap(Map<? extends K,? extends V> m);
// Methods Implementing Map
     public void clear( );
     public boolean containsKey(Object key);
     public boolean containsValue(Object value);
     public Set<Map.Entry<K,V>> entrySet( );
     public boolean equals(Object o);
     public V get(Object key);
     public int hashCode( );
     public boolean isEmpty( );                          default:true
     public Set<K> keySet( );
     public V put(K key, V value);
     public void putAll(Map<? extends K,? extends V> t);
     public V remove(Object key);
     public int size( );
     public Collection<V> values( );
// Public Methods Overriding AbstractMap
     public Object clone( );
}
```

IllegalFormatCodePointException java.util

Java 5.0 ***serializable unchecked***

An `IllegalFormatException` of this type is thrown by a `Formatter` when an `int` used to represent a Unicode character is out of range.

Figure 16-28. java.util.IllegalFormatCodePointException




```
public class IllegalFormatCodePointException extends IllegalFormatException {
// Public Constructors
     public IllegalFormatCodePointException(int c);
// Public Instance Methods
     public int getCodePoint( );
// Public Methods Overriding Throwable
     public String getMessage( );
}
```

IllegalFormatConversionException java.util

Java 5.0 ***serializable unchecked***

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

An `IllegalFormatException` of this type is thrown by a `Formatter` when the type of the `format( )` or `printf( )` argument does not match the type required by the corresponding conversion specifier in the format string.

Figure 16-29. java.util.IllegalFormatConversionException


```
public class IllegalFormatConversionException extends IllegalFormatException {
// Public Constructors
     public IllegalFormatConversionException(char c, Class<?> arg);
// Public Instance Methods
     public Class<?> getArgumentClass( );
     public char getConversion( );
// Public Methods Overriding Throwable
     public String getMessage( );
}
```

IllegalFormatException **java.util**

Java 5.0 ***serializable unchecked***

An exception of this type is thrown by a `Formatter` when there is problem with the format string. This package defines many subclasses of this exception type to describe particular format string problems.

Figure 16-30. java.util.IllegalFormatException


```
public class IllegalFormatException extends IllegalArgumentException {
// No Constructor
}
```

Subclasses

`DuplicateFormatFlagsException`, `FormatFlagsConversionMismatchException`, `IllegalFormatCodePointException`, `IllegalFormatConversionException`, `IllegalFormatFlagsException`, `IllegalFormatPrecisionException`, `IllegalFormatWidthException`, `MissingFormatArgumentException`, `MissingFormatWidthException`, `UnknownFormatConversionException`, `UnknownFormatFlagsException`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

IllegalFormatFlagsException java.util

Java 5.0 ***serializable unchecked***

An `IllegalFormatException` of this type is thrown by a `Formatter` when a format string contains an illegal combination of flags.

Figure 16-31. java.util.IllegalFormatFlagsException


```
public class IllegalFormatFlagsException extends IllegalFormatException {
// Public Constructors
     public IllegalFormatFlagsException(String f);
// Public Instance Methods
     public String getFlags( );
// Public Methods Overriding Throwable
     public String getMessage( );
}
```

IllegalFormatPrecisionException java.util

Java 5.0 ***serializable unchecked***

An `IllegalFormatException` of this type is thrown by a `Formatter` when the precision of a format string is illegal.

Figure 16-32. java.util.IllegalFormatPrecisionException


```
public class IllegalFormatPrecisionException extends IllegalFormatException {
// Public Constructors
     public IllegalFormatPrecisionException(int p);
// Public Instance Methods
     public int getPrecision( );
// Public Methods Overriding Throwable
     public String getMessage( );
}
```

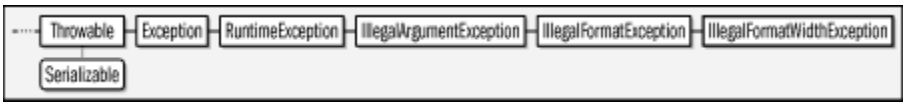

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

IllegalFormatWidthException java.util

Java 5.0 ***serializable unchecked***

An `IllegalFormatException` of this type is thrown by a `Formatter` when the width of a format string is illegal.

Figure 16-33. java.util.IllegalFormatWidthException



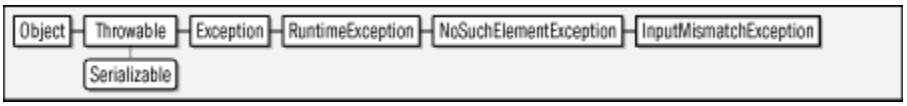

```
public class IllegalFormatWidthException extends IllegalFormatException {
// Public Constructors
     public IllegalFormatWidthException(int w);
// Public Instance Methods
     public int getWidth( );
// Public Methods Overriding Throwable
     public String getMessage( );
}
```

InputMismatchException java.util

Java 5.0 ***serializable unchecked***

An exception of this type is thrown by a `Scanner` that is not of the expected type or is out of range. Note that the `Scanner` implements the `Iterator` interface, and this exception is a subclass of `NoSuchElementException`, which is thrown by `Iterator.next( )` when no more elements are available.

Figure 16-34. java.util.InputMismatchException




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

InvalidPropertiesFormatException java.util

Java 5.0 ***serializable checked***

An exception of this type is thrown by `Properties.loadFromXML( )` if the specified input stream does not contain appropriate XML.

Figure 16-35. java.util.InvalidPropertiesFormatException


```
public class InvalidPropertiesFormatException extends java.io.IOException {
// Public Constructors
     public InvalidPropertiesFormatException(String message);
     public InvalidPropertiesFormatException(Throwable cause);
}
```

Thrown By

`Properties.loadFromXML( )`

Iterator<E> java.util

Java 1.2

This interface defines methods for iterating, or enumerating, the elements of a collection. It has been made generic in Java 5.0 and the type variable *E* represents the type of the elements in the 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( ));
```

In Java 5.0, collections and other classes that can return an `Iterator` implement the `java.lang.Iterable` interface, which allows them to be iterated much more simply with the `for/in` looping statement.

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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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<E> {
// Public Instance Methods
     boolean hasNext( );
     E next( );
     void remove( );
}
```

Implementations

`ListIterator`, `Scanner`

Returned By

Too many methods to list.

LinkedHashMap<K,V> java.util

Java 1.4 ***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 affect 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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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 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.

Figure 16-36. java.util.LinkedHashMap<K,V>


```
public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V> {
// Public Constructors
     public LinkedHashMap( );
     public LinkedHashMap(int initialCapacity);
     public LinkedHashMap(Map<? extends K,? extends V> m);
     public LinkedHashMap(int initialCapacity, float loadFactor);
     public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder);
// Methods Implementing Map
     public void clear( );
     public boolean containsValue(Object value);
     public V get(Object key);
// Protected Instance Methods
     protected boolean removeEldestEntry(Map.Entry<K,V> eldest);     constant
}
```

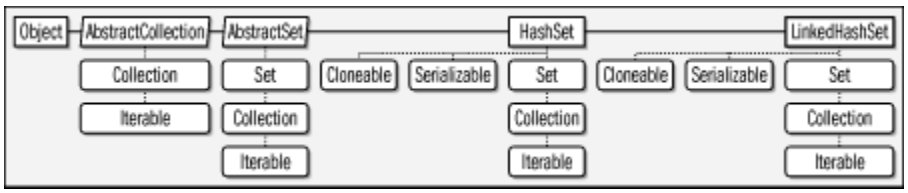
LinkedHashSet<E> **java.util**

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.4 ***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` is used. What is unique about a `LinkedHashSet` is that in addition to the hashtable data structure, it also 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 which they were inserted. By contrast, the elements of a `HashSet` are enumerated in an order that is essentially random. 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.

Figure 16-37. java.util.LinkedHashSet<E>




```
public class LinkedHashSet<E> extends HashSet<E> implements Set<E>, Cloneable, Serializable {
// Public Constructors
     public LinkedHashSet( );
     public LinkedHashSet(Collection<? extends E> c);
     public LinkedHashSet(int initialCapacity);
     public LinkedHashSet(int initialCapacity, float loadFactor);
}
```

LinkedList<E> **java.util**

Java 1.2 ***cloneable serializable collection***

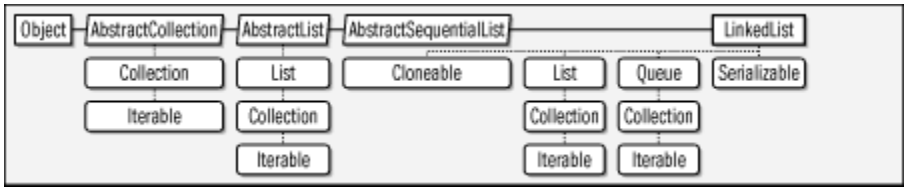
This class implements the `List` interface in terms of a doubly linked list. In Java 5.0, it also implements the `Queue` interface and uses its list as a first-in, first-out (FIFO) queue. `LinkedList` is a generic type, and the type variable *`E`* represents the type of the elements of the list. `LinkedList` supports all optional methods of `List`, `Queue` and `Collection` and allows list elements of any type, including `null` (in this it differs from most `Queue` implementations, which prohibit `null` elements).

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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`.

Figure 16-38. java.util.LinkedList<E>




```
public class LinkedList<E> extends AbstractSequentialList<E>
 implements List<E>, Queue<E>, Cloneable, Serializable {
// Public Constructors
     public LinkedList( );
     public LinkedList(Collection<? extends E> c);
// Public Instance Methods
     public void addFirst(E o);
     public void addLast(E o);
     public E getFirst( );
     public E getLast( );
     public E removeFirst( );
     public E removeLast( );
// Methods Implementing List
     public boolean add(E o);
     public void add(int index, E element);
     public boolean addAll(Collection<? extends E> c);
     public boolean addAll(int index, Collection<? extends E> c);
     public void clear( );
     public boolean contains(Object o);
     public E get(int index);
     public int indexOf(Object o);
     public int lastIndexOf(Object o);
     public ListIterator<E> listIterator(int index);
     public boolean remove(Object o);
     public E remove(int index);
     public E set(int index, E element);
     public int size( );
     public Object[ ] toArray( );
     public <T> T[ ] toArray(T[ ] a);
// Methods Implementing Queue
5.0  public E element( );
5.0  public boolean offer(E o);
5.0  public E peek( );
5.0  public E poll( );
5.0  public E remove( );
// Public Methods Overriding Object
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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

List<E> **java.util**

Java 1.2 ***collection***

This interface represents an ordered collection of objects. In Java 5.0 `List` is a generic interface and the type variable *`E`* represents the type of the objects in the list. 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`* 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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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`, `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`.

Figure 16-39. java.util.List<E>


```
public interface List<E> extends Collection<E> {
// Public Instance Methods
     boolean add(E o);
     void add(int index, E element);
     boolean addAll(Collection<? extends E> c);
     boolean addAll(int index, Collection<? extends E> c);
     void clear( );
     boolean contains(Object o);
     boolean containsAll(Collection<?> c);
     boolean equals(Object o);
     E get(int index);
     int hashCode( );
     int indexOf(Object o);
     boolean isEmpty( );
     Iterator<E> iterator( );
     int lastIndexOf(Object o);
     ListIterator<E> listIterator( );
     ListIterator<E> listIterator(int index);
     boolean remove(Object o);
     E remove(int index);
     boolean removeAll(Collection<?> c);
     boolean retainAll(Collection<?> c);
     E set(int index, E element);
     int size( );
     List<E> subList(int fromIndex, int toIndex);
     Object[ ] toArray( );
     <T> T[ ] toArray(T[ ] a);
}
```

Implementations

`AbstractList`, `ArrayList`, `LinkedList`, `Vector`, `java.util.concurrent.CopyOnWriteArrayList`

Passed To

Too many methods to list.

Returned By

Too many methods to list.

Type Of

`Collections.EMPTY_LIST`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

ListIterator<E> **java.util**

Java 1.2

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.

Like the `Iterator` interface, `ListIterator` has been made generic in Java 5.0. The type variable *`E`* represents the type of the elements on the list.

`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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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`.

Figure 16-40. java.util.ListIterator<E>


```
public interface ListIterator<E> extends Iterator<E> {
// Public Instance Methods
     void add(E o);
     boolean hasNext( );
     boolean hasPrevious( );
     E next( );
     int nextIndex( );
     E previous( );
     int previousIndex( );
     void remove( );
     void set(E o);
}
```

Returned By

`AbstractList.listIterator( )`, `AbstractSequentialList.listIterator( )`,`LinkedList.listIterator( )`, `List.listIterator( )`, `java.util.concurrent.CopyOnWriteArrayList.listIterator( )`

ListResourceBundle java.util

Java 1.1

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`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-41. java.util.ListResourceBundle


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

Locale java.util

Java 1.1 ***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.

Figure 16-42. java.util.Locale


```
public final class Locale implements Cloneable, Serializable {
// Public Constructors
1.4  public Locale(String language);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public Locale(String language, String country);
    public Locale(String language, String country, String variant);
// Public Constants
    public static final Locale CANADA;
    public static final Locale CANADA_FRENCH;
    public static final Locale CHINA;
    public static final Locale CHINESE;
    public static final Locale ENGLISH;
    public static final Locale FRANCE;
    public static final Locale FRENCH;
    public static final Locale GERMAN;
    public static final Locale GERMANY;
    public static final Locale ITALIAN;
    public static final Locale ITALY;
    public static final Locale JAPAN;
    public static final Locale JAPANESE;
    public static final Locale KOREA;
    public static final Locale KOREAN;
    public static final Locale PRC;
    public static final Locale SIMPLIFIED_CHINESE;
    public static final Locale TAIWAN;
    public static final Locale TRADITIONAL_CHINESE;
    public static final Locale UK;
    public static final Locale US;
// Public Class Methods
1.2 public static Locale[ ] getAvailableLocales( );
    public static Locale getDefault( );
1.2 public static String[ ] getISOCountries( );
1.2 public static String[ ] getISOLanguages( );
    public static void setDefault(Locale newLocale);                    synchronized
// Public Instance Methods
    public String getCountry( );
    public final String getDisplayCountry( );
    public String getDisplayCountry(Locale inLocale);
    public final String getDisplayLanguage( );
    public String getDisplayLanguage(Locale inLocale);
    public final String getDisplayName( );
    public String getDisplayName(Locale inLocale);
    public final String getDisplayVariant( );
    public String getDisplayVariant(Locale inLocale);
    public String getISO3Country( ) throws MissingResourceException;
    public String getISO3Language( ) throws MissingResourceException;
    public String getLanguage( );
    public String getVariant( );
// Public Methods Overriding Object
    public Object clone( );
    public boolean equals(Object obj);
    public int hashCode( );
    public final String toString( );
}
```

Passed To

Too many methods to list.

Returned By

java.text.BreakIterator.getAvailableLocales( ), java.text.Collator.getAvailableLocales( ), java.text.DateFormat.getAvailableLocales( ), java.text.MessageFormat.getLocale( ), java.text.NumberFormat.getAvailableLocales( ), Calendar.getAvailableLocales( ),java.util.Formatter.locale( ), ResourceBundle.getLocale( ),Scanner.locale( ), javax.security.auth.callback.LanguageCallback.getLocale( )

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Map<K,V> **java.util**

Java 1.2 ***collection***

This interface represents a collection of mappings, or associations, between key objects and value objects. Hashtables and associative arrays are examples of maps. In Java 5.0 this interface has been made generic. The type variable *K* represents the type of the keys held by the map and the type variable *V* represents the type of the values associated with those keys.

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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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<K,V> {
// Nested Types
     public interface Entry<K,V>;
// Public Instance Methods
     void clear( );
     boolean containsKey(Object key);
     boolean containsValue(Object value);
     Set<Map.Entry<K,V>> entrySet( );
     boolean equals(Object o);
     V get(Object key);
     int hashCode( );
     boolean isEmpty( );
     Set<K> keySet( );
     V put(K key, V value);
     void putAll(Map<? extends K,? extends V> t);
     V remove(Object key);
     int size( );
     Collection<V> values( );
}
```

Implementations

`AbstractMap`, `HashMap`, `Hashtable`, `IdentityHashMap`, `LinkedHashMap`, `SortedMap`, `WeakHashMap`, `java.util.concurrent.ConcurrentMap`, `java.util.jar.Attributes`

Passed To

Too many methods to list.

Returned By

Too many methods to list.

Type Of

`Collections.EMPTY_MAP`, `java.util.jar.Attributes.map`

Map.Entry<K,V> java.util

Java 1.2

This interface represents a single mapping, or association, between a key object and a value object in a `Map`. Like `Map` itself, `Map.Entry` has been made generic in Java 5.0 and defines the same type variables that `Map` does.

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 in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`java.lang.UnsupportedOperationException` if it is not supported by the implementation.

```
public interface Map.Entry<K,V> {
// Public Instance Methods
     boolean equals(Object o);
     K getKey( );
     V getValue( );
     int hashCode( );
     V setValue(V value);
}
```

Passed To

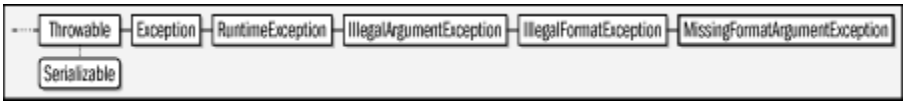
`LinkedHashMap.removeEldestEntry( )`

MissingFormatArgumentException java.util

Java 5.0 ***serializable unchecked***

An `IllegalFormatException` of this type is thrown by a `Formatter` when a `format( )` or `printf( )` method does not have enough arguments to match the number conversion specifiers in the format string.

Figure 16-43. java.util.MissingFormatArgumentException



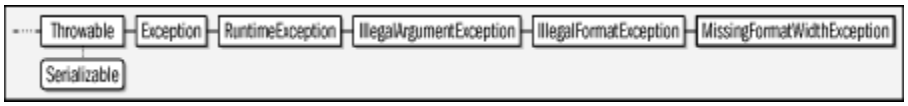

```
public class MissingFormatArgumentException extends IllegalFormatException {
// Public Constructors
     public MissingFormatArgumentException(String s);
// Public Instance Methods
     public String getFormatSpecifier( );
// Public Methods Overriding Throwable
     public String getMessage( );
}
```

MissingFormatWidthException java.util

Java 5.0 ***serializable unchecked***

An `IllegalFormatException` of this type is thrown by a `Formatter` when a format conversion requires a field width, but the width is omitted.

Figure 16-44. java.util.MissingFormatWidthException




```
public class MissingFormatWidthException extends IllegalFormatException {
// Public Constructors
     public MissingFormatWidthException(String s);
// Public Instance Methods
     public String getFormatSpecifier( );
// Public Methods Overriding Throwable
     public String getMessage( );
}
```

MissingResourceException java.util

Java 1.1 ***serializable unchecked***

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.

Figure 16-45. java.util.MissingResourceException


```
public class MissingResourceException extends RuntimeException {
// Public Constructors
     public MissingResourceException(String s, String className, String key);
// Public Instance Methods
     public String getClassName( );
     public String getKey( );
}
```

Thrown By

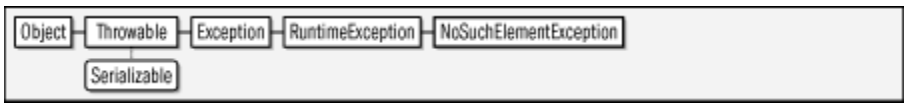
`Locale.{getISO3Country( ),getISO3Language( )}`

NoSuchElementException java.util

Java 1.0 ***serializable unchecked***

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`).

Figure 16-46. java.util.NoSuchElementException




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

Subclasses

`InputMismatchException`

Observable java.util

Java 1.0

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 de-register it by passing it to the `deleteObserver( )` method. You can delete all observers registered for an `Observable` with `deleteObservers( )`, and can 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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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.util

Java 1.0

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
     void update(Observable o, Object arg);
}
```

Passed To

`Observable.{addObserver( ),deleteObserver( )}`

PriorityQueue<E> java.util

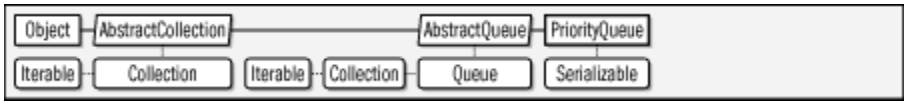
Java 5.0 ***serializable collection***

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

This class is a `Queue` implementation that orders its elements according to a specified `Comparator` or orders `Comparable` elements according to their `compareTo( )` methods. The head of the queue (the element removed by `remove( )` and `poll( )`) is the smallest element on the queue according to this ordering. The `Iterator` return by the `iterator( )` method is not guaranteed to iterate the elements in their sorted order.

`PriorityQueue` is unbounded and prohibits `null` elements. It is not threadsafe.

Figure 16-47. java.util.PriorityQueue<E>




```
public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable {
// Public Constructors
     public PriorityQueue( );
     public PriorityQueue(int initialCapacity);
     public PriorityQueue(SortedSet<? extends E> c);
     public PriorityQueue(PriorityQueue<? extends E> c);
     public PriorityQueue(Collection<? extends E> c);
     public PriorityQueue(int initialCapacity, Comparator<? super E> comparator);
// Public Instance Methods
     public Comparator<? super E> comparator( );
// Methods Implementing Collection
     public Iterator<E> iterator( );
     public boolean remove(Object o);
     public int size( );
// Methods Implementing Queue
     public boolean offer(E o);
     public E peek( );
     public E poll( );
// Public Methods Overriding AbstractQueue
     public boolean add(E o);
     public void clear( );
}
```

Properties java.util

Java 1.0 ***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.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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( )` 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. In Java 5.0, `storeToXML( )` and `loadFromXML( )` are alternatives that write and read properties files using a simple XML grammar.

Figure 16-48. java.util.Properties


```
public class Properties extends Hashtable<Object,Object> {
// 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
5.0  public void loadFromXML(java.io.InputStream in)
        throws java.io.IOException, InvalidPropertiesFormatException;     synchronized
     public Enumeration<?> propertyNames( );
1.2  public Object setProperty(String key, String value);           synchronized
1.2  public void store(java.io.OutputStream out, String comments)
        throws java.io.IOException;     synchronized
5.0  public void storeToXML(java.io.OutputStream os, String comment)
        throws java.io.IOException;     synchronized
5.0  public void storeToXML(java.io.OutputStream os, String comment, String encoding)
throws java.io.IOException;     synchronized
// Protected Instance Fields
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    protected Properties defaults;
// Deprecated Public Methods
#    public void save(java.io.OutputStream out, String comments);     synchronized
}
```

Subclasses

`java.security.Provider`

Passed To

`System.setProperties( )`,
`javax.xml.transform.Transformer.setOutputProperties( )`

Returned By

`System.getProperties( )`,
`javax.xml.transform.Templates.getOutputProperties( )`,
`javax.xml.transform.Transformer.getOutputProperties( )`

PropertyPermission java.util

Java 1.2 ***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.

Figure 16-49. java.util.PropertyPermission

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
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.util

Java 1.1

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`.

Figure 16-50. java.util.PropertyResourceBundle


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

Queue<E> java.util

Java 5.0 ***collection***

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

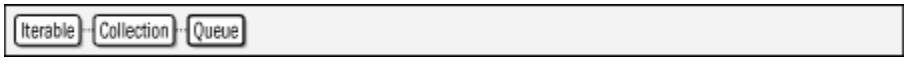
A `Queue<E>` is an ordered `Collection` of elements of type *`E`*. Unlike `List`, the `Queue` interface does not permit indexed access to its elements: elements may be inserted at the *tail* of the queue and may be removed from the *head* of the queue, but the elements in between may not be accessed by their position. Unlike `Set`, `Queue` implementations do not prohibit duplicate elements.

Queues may be manipulated through the methods of the `Collection` interface, including iteration via the `iterator( )` method and the `Iterator` object it returns. It is more common to manipulate queues through the more specialized methods defined by the `Queue` interface, however. Place an element at the tail of the queue with `offer( )`. If the queue is already full, `offer( )` returns `false`. Remove an element from the head of the queue with `remove( )` or `poll( )`. These methods differ only in the case of an empty queue: `remove( )` throws an unchecked `NoSuchElementException` and `poll( )` returns `null`. (Most queue implementations prohibit `null` elements for this reason, but `LinkedList` is an exception.) Query the element at the head of a queue without removing it with `element( )` or `peek( )`. If the queue is empty, `element( )` throws `NoSuchElementException` and `peek( )` returns `null`.

Most `Queue` implementations order their elements in first-in, first-out (FIFO) order. Other implementations may provide other orderings. A queue `Iterator` is not required to traverse the queue's elements in order. A `Queue` implementation with a fixed size is a *bounded* queue. When a bounded queue is full, it is not possible to insert a new element until an element is first removed. Unlike the `List` and `Set` interfaces, the `Queue` interface does not require implementations to override the `equals( )` method, and `Queue` implementations typically do not override it.

In Java 5.0, the `LinkedList` class has been retrofitted to implement `Queue` as well as `List`. `PriorityQueue` is a `Queue` implementation that orders elements based on the `Comparable` or `Comparator` interfaces. `AbstractQueue` is an abstract implementation that offers partial support for simple `Queue` implementations. The `java.util.concurrent` package defines a `BlockingQueue` interface that extends this implementation and includes `Queue` and `BlockingQueue` implementations that are useful in multithreaded programming.

Figure 16-51. java.util.Queue<E>




```
public interface Queue<E> extends Collection<E> {
// Public Instance Methods
     E element( );
     boolean offer(E o);
     E peek( );
     E poll( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    E remove( );
}
```

Implementations

`AbstractQueue`, `LinkedList`, `java.util.concurrent.BlockingQueue`, `java.util.concurrent.ConcurrentLinkedQueue`

Random java.util

Java 1.0 ***serializable***

This class implements a pseudorandom number generator suitable for games and similar applications. If you need a cryptographic-strength source of pseudorandomness, 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. As of Java 1.2, if you pass an argument to `nextInt( )`, it returns a value between zero (inclusive) and the specified number (exclusive). `nextGaussian( )` returns pseudorandom 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 pseudorandom `boolean` value, and `nextBytes( )` fills in the specified `byte` array with pseudorandom bytes. You can use the `setSeed( )` method or the optional constructor argument to initialize the pseudorandom number generator with some variable seed value other than the current time (the default) or with a constant to ensure a repeatable sequence of pseudorandomness.

Figure 16-52. java.util.Random


```
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( );                        synchronized
     public int nextInt( );
1.2  public int nextInt(int n);
     public long nextLong( );
     public void setSeed(long seed);                     synchronized
// Protected Instance Methods
1.1  protected int next(int bits);
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Subclasses

`java.security.SecureRandom`

Passed To

`java.math.BigInteger.{BigInteger( ),probablePrime( )}, Collections.shuffle( )`

RandomAccess java.util

Java 1.4

This marker interface is implemented by `List` implementations to advertise that they provide efficient (usually constant time) random access 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, java.util.concurrent.CopyOnWriteArrayList`

ResourceBundle java.util

Java 1.1

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
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    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<String> 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.util.logging.LogRecord.setResourceBundle( )`

Returned By

`java.util.logging.Logger.getResourceBundle( )`, `java.util.logging.LogRecord.getResourceBundle( )`

Scanner java.util

Java 5.0

This class is a text scanner or tokenizer. It can read input from any `Readable` object, and convenience constructors can read text from a specified string, file, byte stream, or byte channel. The constructors for files, byte streams, and byte channels optionally allow you to specify the name of the charset to use for byte-to-character conversions.

After creating a `Scanner`, you can configure it. `useDelimiter( )` specifies a regular expression (as a `java.util.regex.Pattern` or a `String`) that represents the token delimiter. The default delimiter is any run of whitespace. `useLocale( )` specifies the `Locale` to use for scanning numbers: this may affect things like the character expected for decimal points and the thousands separator. `useRadix( )` specifies the radix, or base, in which numbers should be parsed. Any value between 2 and 36 is allowed. These configuration methods may be called at any time and are not required to be called before scanning begins.

`Scanner` implements the `Iterable<String>` interface, and you can use the `hasNext( )` and `next( )` methods of this interface to break the input into a series of `String` tokens separated by whitespace or by the delimiter specified with

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`useDelimiter( )`. In addition to these `Iterable` methods, however, `Scanner` defines a number of `next`*X* and `hasNext`*X* methods for various numeric types *X*. `nextLine( )` returns the next line of input. Two variants of the `next( )` method accept a regular expression as an argument and return the next chunk of text matching a specified regular expression. The corresponding `hasNext( )` methods accept a regular expression and return `true` if the input matches it.

The `skip( )` method ignores delimiters and skips text matching the specified regular expression. `findInLine( )` looks ahead for text matching the specified regular expression in the current line. If a match is found, the `Scanner` advances past that text and returns it. Otherwise, the `Scanner` returns `null` without advancing. `findWithinHorizon( )` is similar but looks for a match within the specified number of characters (a horizon of 0 specifies an unlimited number).

The `next( )` methods and its `next`*X* variants throw a `NoSuchElementException` if there is no more input text. They throw an `InputMismatchException` (a subclass of `NoSuchElementException`) if the next token cannot be parsed as the specified type or does not match the specified pattern. The `Readable` object that the `Scanner` reads text from may throw a `java.io.IOException`, but, for ease of use, the `Scanner` never propagates this exception. If an `IOException` occurs, the `Scanner` assumes that no more input is available from the `Readable`. Call `ioException( )` to obtain the most recent `IOException`, if any, thrown by the `Readable`.

The `close( )` method checks whether the `Readable` object implements the `Closeable` interface and, if so, calls the `close( )` method on that object. Once `close( )` has been called, any attempt to read tokens from the `Scanner` results in an `IllegalStateException`.

See also `StringTokenizer` and `java.io.StreamTokenizer`.

Figure 16-53. java.util.Scanner


```
public final class Scanner implements Iterator<String> {
// Public Constructors
     public Scanner(Readable source);
     public Scanner(java.nio.channels.ReadableByteChannel source);
     public Scanner(java.io.InputStream source);
     public Scanner(java.io.File source) throws java.io.FileNotFoundException;
     public Scanner(String source);
     public Scanner(java.nio.channels.ReadableByteChannel source, String charsetName);
     public Scanner(java.io.InputStream source, String charsetName);
     public Scanner(java.io.File source, String charsetName)
        throws java.io.FileNotFoundException;
// Public Instance Methods
     public void close( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public java.util.regex.Pattern delimiter( );
    public String findInLine(String pattern);
    public String findInLine(java.util.regex.Pattern pattern);
    public String findWithinHorizon(java.util.regex.Pattern pattern, int horizon);
    public String findWithinHorizon(String pattern, int horizon);
    public boolean hasNext(java.util.regex.Pattern pattern);
    public boolean hasNext(String pattern);
    public boolean hasNextBigDecimal( );
    public boolean hasNextBigInteger( );
    public boolean hasNextBigInteger(int radix);
    public boolean hasNextBoolean( );
    public boolean hasNextByte( );
    public boolean hasNextByte(int radix);
    public boolean hasNextDouble( );
    public boolean hasNextFloat( );
    public boolean hasNextInt( );
    public boolean hasNextInt(int radix);
    public boolean hasNextLine( );
    public boolean hasNextLong( );
    public boolean hasNextLong(int radix);
    public boolean hasNextShort( );
    public boolean hasNextShort(int radix);
    public java.io.IOException ioException( );
    public Locale locale( );
    public java.util.regex.MatchResult match( );
    public String next(String pattern);
    public String next(java.util.regex.Pattern pattern);
    public java.math.BigDecimal nextBigDecimal( );
    public java.math.BigInteger nextBigInteger( );
    public java.math.BigInteger nextBigInteger(int radix);
    public boolean nextBoolean( );
    public byte nextByte( );
    public byte nextByte(int radix);
    public double nextDouble( );
    public float nextFloat( );
    public int nextInt( );
    public int nextInt(int radix);
    public String nextLine( );
    public long nextLong( );
    public long nextLong(int radix);
    public short nextShort( );
    public short nextShort(int radix);
    public int radix( );
    public Scanner skip(java.util.regex.Pattern pattern);
    public Scanner skip(String pattern);
    public Scanner useDelimiter(java.util.regex.Pattern pattern);
    public Scanner useDelimiter(String pattern);
    public Scanner useLocale(Locale locale);
    public Scanner useRadix(int radix);
// Methods Implementing Iterator
    public boolean hasNext( );
    public String next( );
    public void remove( );
// Public Methods Overriding Object
    public String toString( );
}
```

Set<E> java.util

Java 1.2 ***collection***

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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`. In Java 5.0 `Set` is a generic interface and the type variable *`E`* represents the type of the objects in 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 the methods throw `java.lang.UnsupportedOperationException`. See also `Collection`, `List`, `Map`, `SortedSet`, `HashSet`, and `TreeSet`.

Figure 16-54. java.util.Set<E>


```
public interface Set<E> extends Collection<E> {
// Public Instance Methods
     boolean add(E o);
     boolean addAll(Collection<? extends E> c);
     void clear( );
     boolean contains(Object o);
     boolean containsAll(Collection<?> c);
     boolean equals(Object o);
     int hashCode( );
     boolean isEmpty( );
     Iterator<E> iterator( );
     boolean remove(Object o);
     boolean removeAll(Collection<?> c);
     boolean retainAll(Collection<?> c);
     int size( );
     Object[ ] toArray( );
     <T> T[ ] toArray(T[ ] a);
}
```

Implementations

`AbstractSet`, `HashSet`, `LinkedHashSet`, `SortedSet`

Passed To

`java.security.cert.PKIXBuilderParameters.PKIXBuilderParameters( ),java.security.cert.PKIXParameters.{PKIXParameters( ), setInitialPolicies( ),setTrustAnchors( )}, java.security.cert.X509CertSelector.{setExtendedKeyUsage( ), setPolicy( )},java.text.AttributedCharacterIterator.`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
{getRunLimit( ),getRunStart( )},Collections.{checkedSet( ),
synchronizedSet( ),unmodifiableSet( )},
javax.security.auth.Subject.Subject( )
```

Returned By

Too many methods to list.

Type Of

```
Collections.EMPTY_SET
```

SimpleTimeZone java.util

Java 1.1 ***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 nonstandard 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.

Figure 16-55. java.util.SimpleTimeZone


```
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);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
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(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<K,V> java.util

Java 1.2 ***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.

Figure 16-56. java.util.SortedMap<K,V>

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
public interface SortedMap<K,V> extends Map<K,V> {
// Public Instance Methods
     Comparator<? super K> comparator( );
     K firstKey( );
     SortedMap<K,V> headMap(K toKey);
     K lastKey( );
     SortedMap<K,V> subMap(K fromKey, K toKey);
     SortedMap<K,V> tailMap(K fromKey);
}
```

Implementations

`TreeMap`

Passed To

`Collections.{checkedSortedMap( ),synchronizedSortedMap( ), unmodifiableSortedMap( )},TreeMap.TreeMap( )`

Returned By

`java.nio.charset.Charset.availableCharsets( ),Collections. {checkedSortedMap( ),synchronizedSortedMap( ), unmodifiableSortedMap( )},TreeMap.{headMap( ),subMap( ), tailMap( )},java.util.jar.Pack200.Packer.properties( ), java.util.jar.Pack200.Unpacker.properties( )`

SortedSet<E> java.util

Java 1.2 ***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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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.

Figure 16-57. java.util.SortedSet<E>


```
public interface SortedSet<E> extends Set<E> {
// Public Instance Methods
     Comparator<? super E> comparator( );
     E first( );
     SortedSet<E> headSet(E toElement);
     E last( );
     SortedSet<E> subSet(E fromElement, E toElement);
     SortedSet<E> tailSet(E fromElement);
}
```

Implementations

`TreeSet`

Passed To

`Collections.{checkedSortedSet( ),synchronizedSortedSet( ), unmodifiableSortedSet( )},PriorityQueue.PriorityQueue( ), TreeSet.TreeSet( )`

Returned By

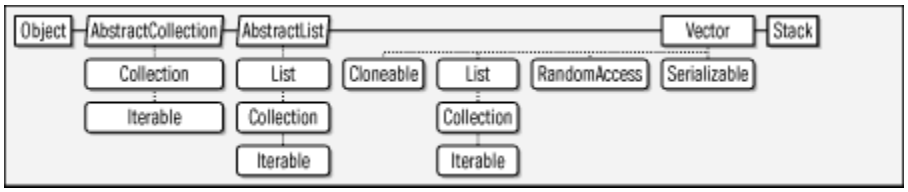
`Collections.{checkedSortedSet( ),synchronizedSortedSet( ), unmodifiableSortedSet( )},TreeSet.{headSet( ),subSet( ), tailSet( )}`

Stack<E> java.util

Java 1.0 ***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.

Figure 16-58. java.util.Stack<E>



Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
public class Stack<E> extends Vector<E> {
// Public Constructors
     public Stack( );
// Public Instance Methods
     public boolean empty( );
     public E peek( );                                    synchronized
     public E pop( );                                     synchronized
     public E push(E item);
     public int search(Object o);                        synchronized
}
```

StringTokenizer java.util

Java 1.0

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( )`.

Figure 16-59. java.util.StringTokenizer


```
public class StringTokenizer implements Enumeration<Object> {
// 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( );
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Timer **java.util**

Java 1.3

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. In Java 5.0 you can also specify the name of the background thread when creating a `Timer`.

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 absolute `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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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);
5.0  public Timer(String name);
5.0  public Timer(String name, boolean isDaemon);
// Public Instance Methods
     public void cancel( );
5.0  public int purge( );
     public void schedule(TimerTask task, long delay);
     public void schedule(TimerTask task, Date time);
     public void schedule(TimerTask task, long delay, long period);
     public void schedule(TimerTask task, Date firstTime, long period);
     public void scheduleAtFixedRate(TimerTask task, long delay, long period);
     public void scheduleAtFixedRate(TimerTask task, Date firstTime, long period);
}
```

TimerTask **java.util**

Java 1.3 ***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.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-60. java.util.TimerTask


```
public abstract class TimerTask implements Runnable {
// Protected Constructors
     protected TimerTask( );
// Public Instance Methods
     public boolean cancel( );
     public long scheduledExecutionTime( );
// Methods Implementing Runnable
     public abstract void run( );
}
```

Passed To

`Timer.{schedule( ),scheduleAtFixedRate( )}`

TimeZone java.util

Java 1.1 ***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.

Figure 16-61. java.util.TimeZone


```
public abstract class TimeZone implements Cloneable, Serializable {
// Public Constructors
     public TimeZone( );
// Public Constants
1.2  public static final int LONG;                                    =1
1.2  public static final int SHORT;                                   =0
// Public Class Methods
     public static String[ ] getAvailableIDs( );                    synchronized
     public static String[ ] getAvailableIDs(int rawOffset);     synchronized
     public static TimeZone getDefault( );                  synchronized
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public static TimeZone getTimeZone(String ID);                    synchronized
    public static void setDefault(TimeZone zone);                synchronized
// Public Instance Methods
1.2 public final String getDisplayName( );
1.2 public final String getDisplayName(Locale locale);
1.2 public final String getDisplayName(boolean daylight, int style);
1.2 public String getDisplayName(boolean daylight, int style, Locale locale);
1.4 public int getDSTSavings( );
    public String getID( );
1.4 public int getOffset(long date);
    public abstract int getOffset(int era, int year, int month, int day,
       int dayOfWeek, int milliseconds);
    public abstract int getRawOffset( );
1.2 public boolean hasSameRules(TimeZone other);
    public abstract boolean inDaylightTime(Date date);
    public void setID(String ID);
    public abstract void setRawOffset(int offsetMillis);
    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( ),setTimeZone( )}, SimpleTimeZone.hasSameRules( ), javax.xml.datatype.XMLGregorianCalendar.toGregorianCalendar( )`

Returned By

`java.text.DateFormat.getTimeZone( ),Calendar.getTimeZone( ), GregorianCalendar.getTimeZone( ), javax.xml.datatype.XMLGregorianCalendar.getTimeZone( )`

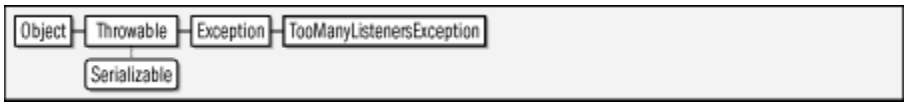
TooManyListenersException java.util

Java 1.1 ***serializable checked***

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.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-62. java.util.TooManyListenersException




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

TreeMap<K,V> java.util

Java 1.2 ***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( )`.

Figure 16-63. java.util.TreeMap<K,V>


```
public class TreeMap<K,V> extends AbstractMap<K,V> implements SortedMap<K,V>,
      Cloneable, Serializable {
// Public Constructors
     public TreeMap( );
     public TreeMap(Comparator<? super K> c);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public TreeMap(SortedMap<K,? extends V> m);
    public TreeMap(Map<? extends K,? extends V> m);
// Methods Implementing Map
    public void clear( );
    public boolean containsKey(Object key);
    public boolean containsValue(Object value);
    public Set<Map.Entry<K,V>> entrySet( );
    public V get(Object key);
    public Set<K> keySet( );
    public V put(K key, V value);
    public void putAll(Map<? extends K,? extends V> map);
    public V remove(Object key);
    public int size( );
    public Collection<V> values( );
// Methods Implementing SortedMap
    public Comparator<? super K> comparator( );
    public K firstKey( );
    public SortedMap<K,V> headMap(K toKey);
    public K lastKey( );
    public SortedMap<K,V> subMap(K fromKey, K toKey);
    public SortedMap<K,V> tailMap(K fromKey);
// Public Methods Overriding AbstractMap
    public Object clone( );
}
```

TreeSet<E> java.util

Java 1.2 ***cloneable serializable collection***

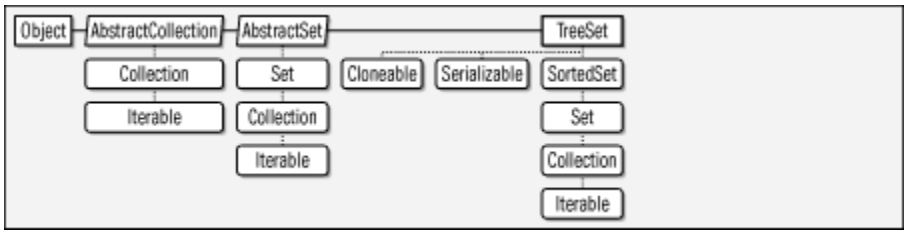
This class implements `SortedSet`, provides support for all optional methods, and guarantees that the elements of the set can be enumerated in ascending order. In order to be sorted, the elements of the set must all be mutually `Comparable` objects, or they must all be compatible with a `Comparator` 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`.

In order 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.

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( )`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-64. java.util.TreeSet<E>




```
public class TreeSet<E> extends AbstractSet<E> implements SortedSet<E>, Cloneable,
       Serializable {
// Public Constructors
     public TreeSet( );
     public TreeSet(Comparator<? super E> c);
     public TreeSet(SortedSet<E> s);
     public TreeSet(Collection<? extends E> c);
// Methods Implementing Set
     public boolean add(E o);
     public boolean addAll(Collection<? extends E> c);
     public void clear( );
     public boolean contains(Object o);
     public boolean isEmpty( );                          default:true
     public Iterator<E> iterator( );
     public boolean remove(Object o);
     public int size( );
// Methods Implementing SortedSet
     public Comparator<? super E> comparator( );
     public E first( );
     public SortedSet<E> headSet(E toElement);
     public E last( );
     public SortedSet<E> subSet(E fromElement, E toElement);
     public SortedSet<E> tailSet(E fromElement);
// Public Methods Overriding Object
     public Object clone( );
}
```

UnknownFormatConversionException java.util

Java 5.0 ***serializable unchecked***

An `IllegalFormatException` of this type is thrown by a `Formatter` when an unknown conversion specifier is included in a format string.

Figure 16-65. java.util.UnknownFormatConversionException


```
public class UnknownFormatConversionException extends IllegalFormatException {
// Public Constructors
     public UnknownFormatConversionException(String s);
// Public Instance Methods
     public String getConversion( );
// Public Methods Overriding Throwable
```

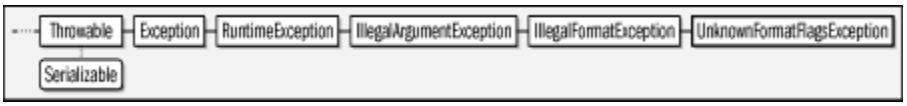
```
    public String getMessage( );
}
```

UnknownFormatFlagsException java.util

Java 5.0 ***serializable unchecked***

An `IllegalFormatException` of this type is thrown by a `Formatter` when unknown flags are specified in a format string.

Figure 16-66. java.util.UnknownFormatFlagsException




```
public class UnknownFormatFlagsException extends IllegalFormatException {
// Public Constructors
    public UnknownFormatFlagsException(String f);
// Public Instance Methods
    public String getFlags( );
// Public Methods Overriding Throwable
    public String getMessage( );
}
```

UUID java.util

Java 5.0 ***serializable comparable***

This class is an immutable representation of 128-bit Universal Unique Identifier, or UUID, which serves as an identifier that is (with very high probability) globally unique. Create a UUID based on random bits with the `randomUUID( )` factory method. Create a UUID based on the MD5 hash code of an array of bytes with the `nameUUIDFromBytes( )` factory method. Or create a UUID by parsing a string with the `fromString( )` factory method. The standard string format of a UUID is 32 hexadecimal digits, broken into five hyphen-separated groups of 8, 4, 4, 4, and 12 digits. For example:

```
7cbf3e1a-d521-40ac-87f1-e28b17530f60
```

Both lowercase and uppercase hex digits are allowed. The `toString( )` method converts a UUID object to a string using this standard format. You can also create a UUID object by explicitly passing the 128 bits in the form of two `long` values to the `UUID( )`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

constructor, but this option should be used only if you are intimately familiar with the relevant UUID standards.

The `toString( )` and `equals( )` methods define the most common operations on a UUID. The `UUID` class implements the `Comparable` interface and defines an ordering for `UUID` objects. Note, however, that the ordering does not represent any meaningful property, such as generation order, of the underlying bits.

Various accessor methods provide details about the bits of a `UUID`, but these details are rarely useful. `getLeastSignificantBits( )` and `getMostSignificantBits( )` return the bits of a UUID as two `long` values. `version( )` and `variant( )` return the version and variant of the UUID, which specify the type (random, name-based, time-based) and bit layout of the UUID. `timestamp( )`, `clockSequence( )`, and `node( )` return values only for time-based UUIDs that have a `version( )` of 1. Note that the `UUID` class does not provide a factory method for creating a time-based UUID.

Figure 16-67. java.util.UUID


```
public final class UUID implements Serializable, Comparable<UUID> {
// Public Constructors
     public UUID(long mostSigBits, long leastSigBits);
// Public Class Methods
     public static UUID fromString(String name);
     public static UUID nameUUIDFromBytes(byte[ ] name);
     public static UUID randomUUID( );
// Public Instance Methods
     public int clockSequence( );
     public long getLeastSignificantBits( );
     public long getMostSignificantBits( );
     public long node( );
     public long timestamp( );
     public int variant( );
     public int version( );
// Methods Implementing Comparable
     public int compareTo(UUID val);
// Public Methods Overriding Object
     public boolean equals(Object obj);
     public int hashCode( );
     public String toString( );
}
```

Vector<E> **java.util**

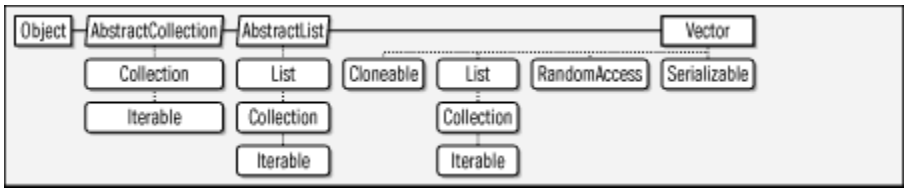
Java 1.0 ***cloneable serializable collection***

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

This class implements an ordered collection—essentially an array—of objects that can grow or shrink as necessary. In Java 1.2, `Vector` has been modified to implement the `List` interface. Unless the `synchronized` methods of the `Vector` class are actually needed, `ArrayList` is preferred in Java 1.2 and later. In Java 5.0 this class has been made generic. The type variable *`E`* represents the type of the elements of the vector.

`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. 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( )`.

Figure 16-68. java.util.Vector<E>




```
public class Vector<E> extends AbstractList<E> implements List<E>,
RandomAccess, Cloneable, Serializable {
// Public Constructors
     public Vector( );
1.2  public Vector(Collection<? extends E> c);
     public Vector(int initialCapacity);
     public Vector(int initialCapacity, int capacityIncrement);
// Public Instance Methods
     public void addElement(E obj);                      synchronized
     public int capacity( );                               synchronized
     public boolean contains(Object elem);           Implements:List
     public void copyInto(Object[ ] anArray);                  synchronized
     public E elementAt(int index);                      synchronized
     public Enumeration<E> elements( );
     public void ensureCapacity(int minCapacity);              synchronized
     public E firstElement( );                           synchronized
     public int indexOf(Object elem);                Implements:List
     public int indexOf(Object elem, int index);               synchronized
     public void insertElementAt(E obj, int index);                  synchronized
     public boolean isEmpty( );                        Implements:List synchronized default:true
     public E 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
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
     public void setElementAt(E 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(E o);                        synchronized
1.2  public void add(int index, E element);
1.2  public boolean addAll(Collection<? extends E> c);      synchronized
1.2  public boolean addAll(int index, Collection<? extends E> 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 E 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 E remove(int index);                         synchronized
1.2  public boolean removeAll(Collection<?> c);             synchronized
1.2  public boolean retainAll(Collection<?> c);             synchronized
1.2  public E set(int index, E element);                      synchronized
     public int size( );                                    synchronized
1.2  public List<E> subList(int fromIndex, int toIndex);     synchronized
1.2  public Object[ ] toArray( );                         synchronized
1.2  public <T> T[ ] toArray(T[ ] a);                    synchronized
// Protected Methods Overriding AbstractList
1.2  protected void removeRange(int fromIndex, int toIndex);        synchronized
// 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`

WeakHashMap<K,V> java.util

Java 1.2 ***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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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 original 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.

Figure 16-69. java.util.WeakHashMap<K,V>


```
public class WeakHashMap<K,V> extends AbstractMap<K,V> implements Map<K,V> {
// Public Constructors
     public WeakHashMap( );
     public WeakHashMap(int initialCapacity);
1.3  public WeakHashMap(Map<? extends K,? extends V> t);
     public WeakHashMap(int initialCapacity, float loadFactor);
// Methods Implementing Map
     public void clear( );
     public boolean containsKey(Object key);
1.4  public boolean containsValue(Object value);
     public Set<Map.Entry<K,V>> entrySet( );
     public V get(Object key);
     public boolean isEmpty( );                          default:true
1.4  public Set<K> keySet( );
     public V put(K key, V value);
1.4  public void putAll(Map<? extends K,? extends V> m);
     public V remove(Object key);
     public int size( );
1.4  public Collection<V> values( );
}
```

Package java.util.concurrent

Java 5.0

This package includes a number of powerful utilities for multithreaded programming. Most of these utilities fall into three main categories:

In addition to these `Map`, `List`, `Set`, and `Queue` implementations, this package also defines the `BlockingQueue` interface. Blocking queues are important in many concurrent algorithms, and this package provides a variety of useful implementations: `ArrayBlockingQueue`, `DelayQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue`, and `SynchronousQueue`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Interfaces

```
public interface BlockingQueue<E> extends java.util.Queue<E>;
public interface Callable<V>;
public interface CompletionService<V>;
public interface ConcurrentMap<K, V> extends java.util.Map<K, V>;
public interface Delayed extends Comparable<Delayed>;
public interface Executor;
public interface ExecutorService extends Executor;
public interface Future<V>;
public interface RejectedExecutionHandler;
public interface ScheduledExecutorService extends ExecutorService;
public interface ScheduledFuture<V> extends Delayed, Future<V>;
public interface ThreadFactory;
```

Enumerated Types

```
public enum TimeUnit;
```

Collections

```
public class ArrayBlockingQueue<E> extends java.util.AbstractQueue<E>
        implements BlockingQueue<E>, Serializable;
public class ConcurrentHashMap<K, V> extends java.util.AbstractMap<K, V>
        implements ConcurrentMap<K, V> Serializable;
public class ConcurrentLinkedQueue<E> extends java.util.AbstractQueue<E>
        implements java.util.Queue<E>, Serializable;
public class CopyOnWriteArrayList<E> implements java.util.List<E>, java.util.RandomAccess, Cloneable, Serializable;
public class CopyOnWriteArraySet<E> extends java.util.AbstractSet<E>
        implements Serializable;
public class DelayQueue<E extends Delayed> extends java.util.AbstractQueue<E>
        implements BlockingQueue<E>;
public class LinkedBlockingQueue<E> extends java.util.AbstractQueue<E>
        implements BlockingQueue<E>, Serializable;
public class PriorityBlockingQueue<E> extends java.util.AbstractQueue<E>
        implements BlockingQueue<E>, Serializable;
public class SynchronousQueue<E> extends java.util.AbstractQueue<E>
        implements BlockingQueue<E>, Serializable;
```

Other Classes

```
public abstract class AbstractExecutorService implements ExecutorService;
   public class ThreadPoolExecutor extends AbstractExecutorService;
      public class ScheduledThreadPoolExecutor extends ThreadPoolExecutor
         implements ScheduledExecutorService;
public class CountDownLatch;
public class CyclicBarrier;
public class Exchanger<V>;
public class ExecutorCompletionService<V> implements CompletionService<V>;
public class Executors;
public class FutureTask<V> implements Future<V>, Runnable;
public class Semaphore implements Serializable;
public static class ThreadPoolExecutor.AbortPolicy implements RejectedExecutionHandler;
public static class ThreadPoolExecutor.CallerRunsPolicy implements RejectedExecutionHandler;
public static class ThreadPoolExecutor.DiscardOldestPolicy implements RejectedExecutionHandler;
public static class ThreadPoolExecutor.DiscardPolicy implements RejectedExecutionHandler;
```

Exceptions

```
public class BrokenBarrierException extends Exception;
public class CancellationException extends IllegalStateException;
public class ExecutionException extends Exception;
public class RejectedExecutionException extends RuntimeException;
public class TimeoutException extends Exception;
```

AbstractExecutorService java.util.concurrent

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 5.0

This abstract class implements the `submit( )`, `invokeAll( )`, and `invokeAny( )` methods of the `ExecutorService` interface. It does not implement the `ExecutorService` shutdown methods or the crucial `execute( )` method for asynchronous execution of `Runnable` tasks.

The methods implemented by `AbstractExecutorService` wrap the submitted `Callable` or `Runnable` task in a `FutureTask` object. `FutureTask` implements `Runnable` and `Future`, which are first passed to the abstract `execute( )` method to be run asynchronously and then returned to the caller.

See `ThreadPoolExecutor` for a concrete implementation, and see `Executors` for convenient `ExecutorService` factory methods.

Figure 16-70. java.util.concurrent.AbstractExecutorService


```
public abstract class AbstractExecutorService implements ExecutorService {
// Public Constructors
     public AbstractExecutorService( );
// Methods Implementing ExecutorService
     public <T> java.util.List<Future<T>> invokeAll(java.util.Collection<Callable<T>> tasks)
        throws InterruptedException;
     public <T> java.util.List<Future<T>> invokeAll(java.util.Collection<Callable<T>> tasks,
        long timeout, TimeUnit unit) throws InterruptedException;
     public <T> T invokeAny(java.util.Collection<Callable<T>> tasks)
        throws InterruptedException, ExecutionException;
     public <T> T invokeAny(java.util.Collection<Callable<T>> tasks, long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
     public Future<?> submit(Runnable task);
     public <T> Future<T> submit(Callable<T> task);
     public <T> Future<T> submit(Runnable task, T result);
}
```

Subclasses

`ThreadPoolExecutor`

ArrayBlockingQueue<E> — java.util.concurrent

Java 5.0 — ***serializable collection***

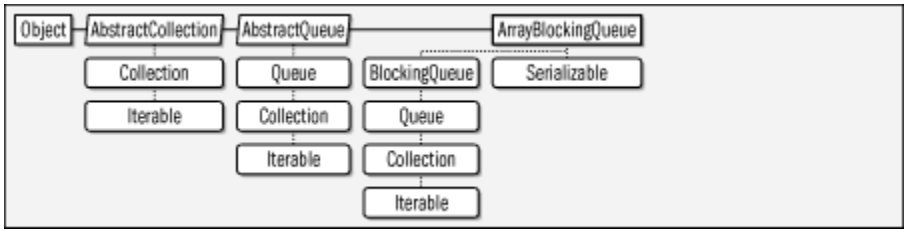
This `BlockingQueue` implementation uses an array to store queue elements. The internal array has a fixed size that is specified when the queue is created, which means that

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

this is a bounded queue and the `put( )` method blocks when the queue has no more room. `ArrayBlockingQueue` orders its elements on a first-in, first-out (FIFO) basis. As with all `BlockingQueue` implementations, `null` elements are prohibited.

If you pass `true` as the second argument to the `ArrayBlockingQueue` constructor, the queue enforces a fairness policy for blocked threads: threads blocked in `put( )` or `take( )` are themselves queued in FIFO order, and the thread that has been waiting the longest is served first. This prevents thread starvation but may decrease overall throughput for the `ArrayBlockingQueue`.

Figure 16-71. java.util.concurrent.ArrayBlockingQueue<E>




```
public class ArrayBlockingQueue<E> extends java.util.AbstractQueue<E>
        implements BlockingQueue<E>, Serializable {
// Public Constructors
     public ArrayBlockingQueue(int capacity);
     public ArrayBlockingQueue(int capacity, boolean fair);
     public ArrayBlockingQueue(int capacity, boolean fair, java.util.Collection<? extends E> c);
// Methods Implementing BlockingQueue
     public int drainTo(java.util.Collection<? super E> c);
     public int drainTo(java.util.Collection<? super E> c, int maxElements);
     public boolean offer(E o);
     public boolean offer(E o, long timeout, TimeUnit unit) throws InterruptedException;
     public E poll(long timeout, TimeUnit unit) throws InterruptedException;
     public void put(E o) throws InterruptedException;
     public int remainingCapacity( );
     public E take( ) throws InterruptedException;
// Methods Implementing Collection
     public void clear( );
     public boolean contains(Object o);
     public java.util.Iterator<E> iterator( );
     public boolean remove(Object o);
     public int size( );
     public Object[ ] toArray( );
     public <T> T[ ] toArray(T[ ] a);
// Methods Implementing Queue
     public E peek( );
     public E poll( );
// Public Methods Overriding AbstractCollection
     public String toString( );
}
```

BlockingQueue<E> **java.util.concurrent**

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 5.0 ***collection***

This interface extends the `java.util.Queue` interface of the Java Collections Framework and adds blocking `put( )` and `take( )` methods. Blocking queues are useful in many concurrent algorithms in which a producer thread puts objects onto a queue and a consumer thread removes them for some kind of processing. The producer thread must block if a bounded queue fills up, and the consumer thread must block if no objects are available on the queue.

In addition to `put( )` and `take( )` methods that block indefinitely, `BlockingQueue` also defines timed versions of the `Queue` methods `offer( )` and `poll( )` that wait up to the specified time. The timeout is specified as both a `long` and a `TimeUnit` constant.

`drainTo( )` removes all available elements from a `BlockingQueue`, adds them to the specified collection, and returns the number of elements removed from the queue. `drainTo( )` does not block. A variant on this method puts an upper bound on the number of elements removed from the queue.

`remainingCapacity( )` returns the number of elements that can be added to the queue before it becomes full or returns `Integer.MAX_VALUE` if the `BlockingQueue` is not a bounded queue. For bounded queues, this method provides a hint as to whether a call to `put( )` will block.

`BlockingQueue` implementations are not allowed to accept `null` elements. The `BlockingQueue` interface refines the `Collection.add( )` and `Queue.offer( )` contracts to indicate that these methods throw `NullPointerException` if passed a `null` value.

Figure 16-72. java.util.concurrent.BlockingQueue<E>


```
public interface BlockingQueue<E> extends java.util.Queue<E> {
// Public Instance Methods
     boolean add(E o);
     int drainTo(java.util.Collection<? super E> c);
     int drainTo(java.util.Collection<? super E> c, int maxElements);
     boolean offer(E o);
     boolean offer(E o, long timeout, TimeUnit unit) throws InterruptedException;
     E poll(long timeout, TimeUnit unit) throws InterruptedException;
     void put(E o) throws InterruptedException;
     int remainingCapacity( );
     E take( ) throws InterruptedException;
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Implementations

`ArrayBlockingQueue`, `DelayQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue`, `SynchronousQueue`

Passed To

`ExecutorCompletionService.ExecutorCompletionService( )`, `ThreadPoolExecutor.ThreadPoolExecutor( )`

Returned By

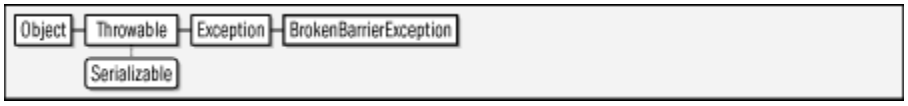
`ScheduledThreadPoolExecutor.getQueue( )`, `ThreadPoolExecutor.getQueue( )`

BrokenBarrierException — java.util.concurrent

Java 5.0 — ***serializable checked***

An exception of this type is thrown when a thread calls `CyclicBarrier.await( )` on a broken barrier, or when the barrier is broken while a thread is waiting. A `CyclicBarrier` enters a broken state when one of the waiting threads is interrupted or times out.

Figure 16-73. java.util.concurrent.BrokenBarrierException




```
public class BrokenBarrierException extends Exception {
// Public Constructors
     public BrokenBarrierException( );
     public BrokenBarrierException(String message);
}
```

Thrown By

`CyclicBarrier.await( )`

Callable<V> — java.util.concurrent

Java 5.0

This interface is a generalized form of the `java.lang.Runnable` interface. Unlike the `run( )` method of `Runnable`, the `call( )` method of `Callable` can return a value and throw an `Exception`. `Callable` is a generic type, and the type variable *`V`* represents the return type of the `call( )` method.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

An `ExecutorService` accepts `Callable` objects for asynchronous execution and returns a `Future` object representing the future result of the `call( )` method.

```
public interface Callable<V> {
// Public Instance Methods
     V call( ) throws Exception;
}
```

Passed To

`AbstractExecutorService.submit( ),CompletionService.submit( ), ExecutorCompletionService.submit( ),Executors. {privilegedCallable( ), privilegedCallableUsingCurrentClassLoader( )}, ExecutorService.submit( ),FutureTask.FutureTask( ), ScheduledExecutorService.schedule( ),ScheduledThreadPoolExecutor. {schedule( ),submit( )}`

Returned By

`Executors.{callable( ),privilegedCallable( ), privilegedCallableUsingCurrentClassLoader( )}`

CancellationException java.util.concurrent

Java 5.0 ***serializable unchecked***

An exception of this type is thrown to indicate that the result of a computation cannot be retrieved because the computation was canceled. The `get( )` method of the `Future` interface may throw a `CancellationException`, for example.

Figure 16-74. java.util.concurrent.CancellationException


```
public class CancellationException extends IllegalStateException {
// Public Constructors
     public CancellationException( );
     public CancellationException(String message);
}
```

CompletionService<V> java.util.concurrent

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 5.0

This interface combines the features of an `ExecutorService` with the features of a `BlockingQueue`. A producer thread may submit `Callable` or `Runnable` tasks for asynchronous execution. As each submitted task completes, its result, in the form of a `Future` object, becomes available to be removed from the queue by a consumer thread that calls `poll( )` or `take( )`.

This generic type declares a type variable *`V`*, which represents the result type of all tasks on the queue.

```
public interface CompletionService<V> {
// Public Instance Methods
     Future<V> poll( );
     Future<V> poll(long timeout, TimeUnit unit) throws InterruptedException;
     Future<V> submit(Callable<V> task);
     Future<V> submit(Runnable task, V result);
     Future<V> take( ) throws InterruptedException;
}
```

Implementations

`ExecutorCompletionService`

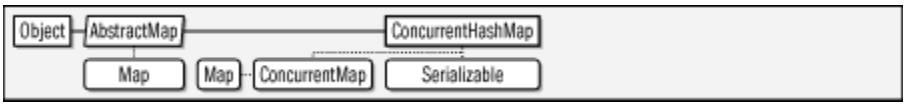
ConcurrentHashMap<K,V> java.util.concurrent

Java 5.0 ***serializable collection***

This class is a threadsafe implementation of the `java.util.Map` interface, and of the atomic operations added by the `ConcurrentMap` interface. This class is intended as a drop-in replacement for `java.util.Hashtable`. It is more efficient than that class, however, because it provides threadsafety without using `synchronized` methods that lock the entire data structure. `ConcurrentHashMap` allows any number of concurrent read operations without locking. Locking is required for updates to a `ConcurrentHashMap`, but the internal data structure is segmented so that only the segment being updated is locked, and reads and writes can proceed concurrently in other segments. You can specify the number of internal segments with the *`concurrencyLevel`* argument to the constructor. The default is 16. Set this to the approximate number of updater threads you expect to access the data structure. Like `Hashtable`, `ConcurrentHashMap` does not allow `null` keys or values. (Note that this differs from the behavior of `java.util.HashMap`.)

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-75. java.util.concurrent.ConcurrentHashMap<K,V>




```
public class ConcurrentHashMap<K,V> extends java.util.AbstractMap<K,V>
 implements ConcurrentMap<K,V>, Serializable {
// Public Constructors
     public ConcurrentHashMap( );
     public ConcurrentHashMap(java.util.Map<? extends K,? extends V> t);
     public ConcurrentHashMap(int initialCapacity);
     public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel);
// Public Instance Methods
     public boolean contains(Object value);
     public java.util.Enumeration<V> elements( );
     public java.util.Enumeration<K> keys( );
// Methods Implementing ConcurrentMap
     public V putIfAbsent(K key, V value);
     public boolean remove(Object key, Object value);
     public V replace(K key, V value);
     public boolean replace(K key, V oldValue, V newValue);
// Methods Implementing Map
     public void clear( );
     public boolean containsKey(Object key);
     public boolean containsValue(Object value);
     public java.util.Set<java.util.Map.Entry<K,V>> entrySet( );
     public V get(Object key);
     public boolean isEmpty( );                          default:true
     public java.util.Set<K> keySet( );
     public V put(K key, V value);
     public void putAll(java.util.Map<? extends K,? extends V> t);
     public V remove(Object key);
     public int size( );
     public java.util.Collection<V> values( );
}
```

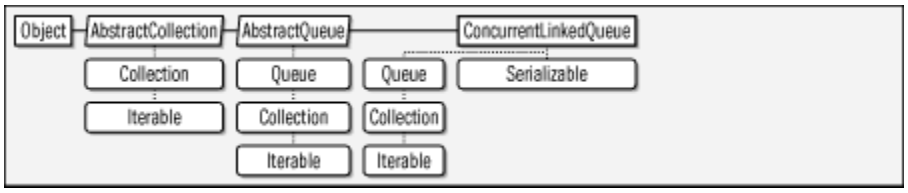
ConcurrentLinkedQueue<E> java.util.concurrent

Java 5.0 ***serializable collection***

This class is a threadsafe implementation of the `java.util.Queue` interface (but not of the `BlockingQueue` interface). It provides threadsafety without using `synchronized` methods that would lock the entire data structure. `ConcurrentLinkedQueue` is unbounded and orders its elements on a first-in, first-out (FIFO) basis. `null` elements are not allowed. This implementation uses a linked-list data structure internally. Note that the `size( )` method must traverse the internal data structure and is therefore a relatively expensive operation for this class.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-76. java.util.concurrent.ConcurrentLinkedQueue<E>




```
public class ConcurrentLinkedQueue<E> extends java.util.AbstractQueue<E>
 implements java.util.Queue<E>, Serializable {
// Public Constructors
     public ConcurrentLinkedQueue( );
     public ConcurrentLinkedQueue(java.util.Collection<? extends E> c);
// Methods Implementing Collection
     public boolean add(E o);
     public boolean contains(Object o);
     public boolean isEmpty( );                          default:true
     public java.util.Iterator<E> iterator( );
     public boolean remove(Object o);
     public int size( );
     public Object[ ] toArray( );
     public <T> T[ ] toArray(T[ ] a);
// Methods Implementing Queue
     public boolean offer(E o);
     public E peek( );
     public E poll( );
}
```

ConcurrentMap<K,V> java.util.concurrent

Java 5.0 ***collection***

This interface extends the `java.util.Map` interface to add four important atomic methods. As with the `Map` interface, the type variables *`K`* and *`V`* represent the types of the mapped keys and values.

`putIfAbsent( )` atomically tests whether a key is already defined in the map, and if not, maps it to the specified value. `remove( )` atomically removes the specified key from the map, but only if it is mapped to the specified value. It returns `true` if it modified the map. There are two versions of the atomic `replace( )` method. The first checks whether the specified value is already mapped to a value. If so, it replaces the existing mapping with the specified value and returns `true`. Otherwise, it returns `false`. The three-argument version of `replace( )` maps the specified key to the specified new value, but only if the key is currently mapped to the specified old value. It returns `true` if the replacement was made and `false` otherwise.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-77. java.util.concurrent.ConcurrentMap<K,V>


```
public interface ConcurrentMap<K,V> extends java.util.Map<K,V> {
// Public Instance Methods
     V putIfAbsent(K key, V value);
     boolean remove(Object key, Object value);
     V replace(K key, V value);
     boolean replace(K key, V oldValue, V newValue);
}
```

Implementations

`ConcurrentHashMap`

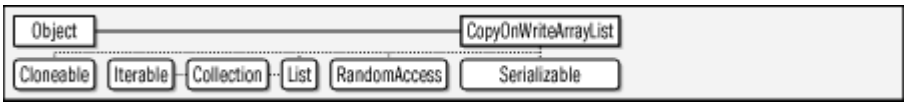
CopyOnWriteArrayList<E> java.util.concurrent

Java 5.0 ***cloneable serializable collection***

This class is a threadsafe `java.util.List` implementation based on an array. Any number of read operations may proceed concurrently. All update methods are `synchronized` and make a completely new copy of the internal array, so this class is best suited to applications in which reads greatly outnumber updates. The `Iterator` of a `CopyOnWriteArrayList` operates on the copy of the array that was current when the `iterator( )` method was called: it does not see any updates that occur after the call to `iterator( )` and is guaranteed never to throw `ConcurrentModificationException`. Update methods of the `Iterator` and `ListIterator` interfaces are not supported and throw `UnsupportedOperationException`.

`CopyOnWriteArrayList` defines a few useful methods beyond those specified by the `List` interface. `addIfAbsent( )` atomically adds an element to the list, but only if the list does not already contain that element. `addAllAbsent( )` adds all elements of a collection that are not already in the list. Two new `indexOf( )` and `lastIndexOf( )` methods are defined that specify a starting index for the search. These provide a convenient alternative to using a `subList( )` view when searching for repeated matches in a list.

Figure 16-78. java.util.concurrent.CopyOnWriteArrayList<E>



Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
public class CopyOnWriteArrayList<E> implements java.util.List<E>,
        java.util.RandomAccess, Cloneable, Serializable {
// Public Constructors
     public CopyOnWriteArrayList( );
     public CopyOnWriteArrayList(java.util.Collection<? extends E> c);
     public CopyOnWriteArrayList(E[ ] toCopyIn);
// Public Instance Methods
     public int addAllAbsent(java.util.Collection<? extends E> c);     synchronized
     public boolean addIfAbsent(E element);                          synchronized
     public int indexOf(E elem, int index);
     public int lastIndexOf(E elem, int index);
// Methods Implementing List
     public boolean add(E element);                        synchronized
     public void add(int index, E element);                          synchronized
     public boolean addAll(java.util.Collection<? extends E> c);     synchronized
     public boolean addAll(int index, java.util.Collection<? extends E> c);     synchronized
     public void clear( );                                  synchronized
     public boolean contains(Object elem);
     public boolean containsAll(java.util.Collection<?> c);
     public boolean equals(Object o);
     public E get(int index);
     public int hashCode( );
     public int indexOf(Object elem);
     public boolean isEmpty( );                            default:true
     public java.util.Iterator<E> iterator( );
     public int lastIndexOf(Object elem);
     public java.util.ListIterator<E> listIterator( );
     public java.util.ListIterator<E> listIterator(int index);
     public boolean remove(Object o);                      synchronized
     public E remove(int index);                           synchronized
     public boolean removeAll(java.util.Collection<?> c);       synchronized
     public boolean retainAll(java.util.Collection<?> c);       synchronized
     public E set(int index, E element);                   synchronized
     public int size( );
     public java.util.List<E> subList(int fromIndex, int toIndex);     synchronized
     public Object[ ] toArray( );
     public <T> T[ ] toArray(T[ ] a);
// Public Methods Overriding Object
     public Object clone( );
     public String toString( );
}
```

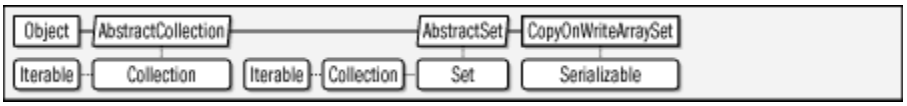
CopyOnWriteArraySet<E> **java.util.concurrent**

Java 5.0 ***serializable collection***

This class is a threadsafe `java.util.Set` implementation based on the `CopyOnWriteArrayList` class. Because the data structure is array-based, the `contains( )` method is `O(n)`; this means that this class is suitable only for relatively small sets. Because the data structure uses copy-on-write, the class is best suited to cases where read operations and traversals greatly outnumber update operations. Iteration over the members of the set is efficient, and the `Iterator` returned by `iterator( )` never throws `ConcurrentModificationException`. The `remove( )` method of the iterator throws `UnsupportedOperationException`. See also `CopyOnWriteArrayList`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-79. java.util.concurrent.CopyOnWriteArraySet<E>




```
public class CopyOnWriteArraySet<E> extends java.util.AbstractSet<E> implements Serializable {
// Public Constructors
     public CopyOnWriteArraySet( );
     public CopyOnWriteArraySet(java.util.Collection<? extends E> c);
// Methods Implementing Set
     public boolean add(E o);
     public boolean addAll(java.util.Collection<? extends E> c);
     public void clear( );
     public boolean contains(Object o);
     public boolean containsAll(java.util.Collection<?> c);
     public boolean isEmpty( );                          default:true
     public java.util.Iterator<E> iterator( );
     public boolean remove(Object o);
     public boolean retainAll(java.util.Collection<?> c);
     public int size( );
     public Object[ ] toArray( );
     public <T> T[ ] toArray(T[ ] a);
// Public Methods Overriding AbstractSet
     public boolean removeAll(java.util.Collection<?> c);
}
```

CountDownLatch java.util.concurrent

Java 5.0

This class synchronizes threads. All threads that call `await( )` block until the `countDown( )` method is invoked a specified number of times. The required number of calls is specified when the `CountDownLatch` is created. Once `countDown( )` has been called the required number of times, all threads blocked in `await( )` are allowed to resume, and any subsequent calls to `await( )` do not block. `getCount( )` returns the number of calls to `countDown( )` that must still be made before the threads blocked in `await( )` can resume. Note that there is no way to reset the count. Once a `CountDownLatch` has "latched," it remains in that state forever. Create a new `CountDownLatch` if you need to synchronize another group of threads. Contrast this class with `CyclicBarrier`.

```
public class CountDownLatch {
// Public Constructors
     public CountDownLatch(int count);
// Public Instance Methods
     public void await( ) throws InterruptedException;
     public boolean await(long timeout, TimeUnit unit) throws InterruptedException;
     public void countDown( );
     public long getCount( );
// Public Methods Overriding Object
     public String toString( );
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

CyclicBarrier java.util.concurrent

Java 5.0

This class synchronizes a group of *n* threads, where *n* is specified to the `CyclicBarrier( )` constructor. Threads call the `await( )` method, which blocks until *n* threads are waiting. In the simple case, all *n* threads are then allowed to proceed, and the `CyclicBarrier` resets itself until it has another *n* threads blocked in `await( )`.

More complex behavior is possible if you pass a `Runnable` object to the `CyclicBarrier` constructor. This `Runnable` is a "barrier action" and when the last of the *n* threads invokes `await( )`, that method uses the thread to invoke the `run( )` method of the `Runnable`. This `Runnable` is typically used to perform some sort of coordinating action on the blocked threads. When the `run( )` method returns, the `CyclicBarrier` allows all blocked threads to resume.

When threads resume from `await( )`, the return value of `await( )` is an integer that represents the order in which they called `await( )`. This is useful if you want to be able to distinguish between otherwise identical worker threads. For example, you might have the thread that arrived first perform some special action while the remaining threads resume.

If any thread times out or is interrupted while blocked in `await( )`, the `CyclicBarrier` is said to be "broken," and all waiting threads (and any threads that subsequently call `await( )`) wake up with a `BrokenBarrierException`. Waiting threads also receive a `BrokenBarrierException` if the `CyclicBarrier` is `reset( )`. The `reset( )` method is the only way to restore a broken barrier to its initial state. This is difficult to coordinate properly, however, unless one controller thread is coded differently from the other threads at the barrier.

```
public class CyclicBarrier {
// Public Constructors
     public CyclicBarrier(int parties);
     public CyclicBarrier(int parties, Runnable barrierAction);
// Public Instance Methods
     public int await( ) throws InterruptedException, BrokenBarrierException;
     public int await(long timeout, TimeUnit unit)
        throws InterruptedException, BrokenBarrierException, TimeoutException;
     public int getNumberWaiting( );
     public int getParties( );
     public boolean isBroken( );
     public void reset( );
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Delayed java.util.concurrent

Java 5.0 ***comparable***

An object that implements this interface has an associated delay. Typically, it is some kind of task, such as a `Callable`, that has been scheduled to execute at some future time. `getDelay( )` returns the remaining time, measured in the specified `TimeUnit`. If no time remains, `getDelay( )` should return zero or a negative value. See `ScheduledFuture` and `DelayQueue`.

Figure 16-80. java.util.concurrent.Delayed


```
public interface Delayed extends Comparable<Delayed> {
// Public Instance Methods
     long getDelay(TimeUnit unit);
}
```

Implementations

`ScheduledFuture`

Passed To

`DelayQueue.{add( ),offer( ),put( )}`

Returned By

`DelayQueue.{peek( ),poll( ),take( )}`

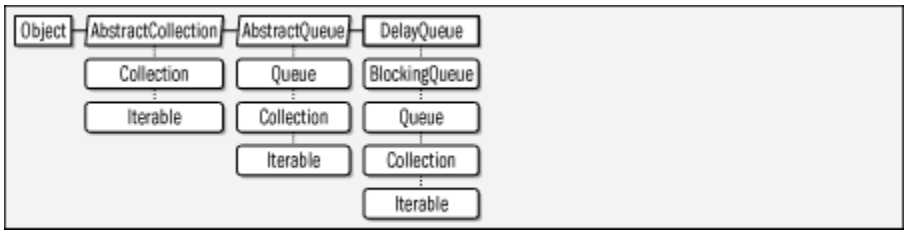
DelayQueue<E extends Delayed> java.util.concurrent

Java 5.0 ***collection***

This `BlockingQueue` implementation restricts its elements to instances of some class *`E`* that implements the `Delay` interface. `null` elements are not allowed. Elements on the queue are ordered by the amount of delay remaining. The element whose `getDelay( )` method returns the smallest value is the first to be removed from the queue. No element may be removed, however, until its `getDelay( )` method returns zero or a negative number.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-81. java.util.concurrent.DelayQueue<E extends Delayed>




```
public class DelayQueue<E extends Delayed> extends java.util.AbstractQueue<E>
        implements BlockingQueue<E> {
// Public Constructors
     public DelayQueue( );
     public DelayQueue(java.util.Collection<? extends E> c);
// Public Instance Methods
     public E peek( );
     public E poll( );
// Methods Implementing BlockingQueue
     public boolean add(E o);
     public int drainTo(java.util.Collection<? super E> c);
     public int drainTo(java.util.Collection<? super E> c, int maxElements);
     public boolean offer(E o);
     public boolean offer(E o, long timeout, TimeUnit unit);
     public E poll(long timeout, TimeUnit unit) throws InterruptedException;
     public void put(E o);
     public int remainingCapacity( );
     public E take( ) throws InterruptedException;
// Methods Implementing Collection
     public void clear( );
     public java.util.Iterator<E> iterator( );
     public boolean remove(Object o);
     public int size( );
     public Object[ ] toArray( );
     public <T> T[ ] toArray(T[ ] array);
}
```

Exchanger<V> java.util.concurrent

Java 5.0

This class allows two threads to rendezvous and exchange data. This is a generic type, and the type variable `V` represents the type of data to be exchanged. Each thread should call `exchange( )` and pass the value of type `V` that it wants to exchange. The first thread to call `exchange( )` blocks until the second thread calls it. At that point, both threads resume. Both threads receive as their return value the object of type `V` passed by the other thread. Note that this class also defines a timed version of `exchange( )` that throws a `TimeoutException` if no exchange occurs within the specified timeout interval. Unlike a `CountDownLatch`, which is a one-shot latch, and `CyclicBarrier` which can be "broken," an `Exchanger` may be reused for any number of exchanges.

```
public class Exchanger<V> {
// Public Constructors
     public Exchanger( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
// Public Instance Methods
     public V exchange(V x) throws InterruptedException;
     public V exchange(V x, long timeout, TimeUnit unit)
        throws InterruptedException, TimeoutException;
}
```

ExecutionException java.util.concurrent

Java 5.0 ***serializable checked***

An exception of this type is like a checked wrapper around an arbitrary exception thrown while executing a task. The `get( )` method of a `Future` object, for example, throws an `ExecutionException` if the `call( )` method of a `Callable` throws an exception. `ExecutionException` may also be thrown by `ExecutorService.invokeAny( )`. Use the `Throwable.getCause( )` method to obtain the exception object that the `ExecutionException` wraps.

Figure 16-82. java.util.concurrent.ExecutionException


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

Thrown By

`AbstractExecutorService.invokeAny( )`, `ExecutorService.invokeAny( )`, `Future.get( )`, `FutureTask.get( )`

Executor java.util.concurrent

Java 5.0

This interface defines a mechanism for executing `Runnable` tasks. A variety of implementations are possible for the `execute( )` method. An implementation might simply synchronously invoke the `run( )` method of the specified `Runnable`. Another

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

implementation might create and start a new thread for each `Runnable` object it is passed. Another might select an existing thread from a thread pool to run the `Runnable` or queue the `Runnable` for future execution when a thread becomes available.

`ExecutorService` extends this interface with methods to execute `Callable` tasks and methods for canceling tasks. `ThreadPoolExecutor` is an `ExecutorService` implementation that creates a configurable thread pool. Finally, the `Executors` class defines a number of factory methods for easily obtaining `ExecutorService` instances.

```
public interface Executor {
// Public Instance Methods
     void execute(Runnable command);
}
```

Implementations

`ExecutorService`

Passed To

`ExecutorCompletionService.ExecutorCompletionService( )`

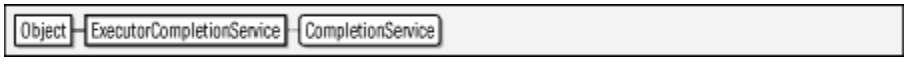
ExecutorCompletionService<V> java.util.concurrent

Java 5.0

This class implements the `CompletionService` interface, which uses an `Executor` object passed to its constructor for executing the tasks passed to its `submit( )` method. As these tasks complete, their result (or exception) is placed, in the form of a `Future` object, on an internal queue and becomes available for removal with the blocking `take( )` method or the nonblocking or timed `poll( )` methods.

This class is useful when you want to execute a number of tasks concurrently and want to process their results in whatever order they complete. See `Executors` for a source of `Executor` objects to use with this class.

Figure 16-83. java.util.concurrent.ExecutorCompletionService<V>




```
public class ExecutorCompletionService<V> implements CompletionService<V> {
// Public Constructors
     public ExecutorCompletionService(Executor executor);
     public ExecutorCompletionService(Executor executor, BlockingQueue<Future<V>>
        completionQueue);
// Methods Implementing CompletionService
     public Future<V> poll( );
     public Future<V> poll(long timeout, TimeUnit unit) throws InterruptedException;
     public Future<V> submit(Callable<V> task);
     public Future<V> submit(Runnable task, V result);
```

```
    public Future<V> take( ) throws InterruptedException;
}
```

Executors **java.util.concurrent**

Java 5.0

This utility class defines static factory methods for creating `ExecutorService` and `ScheduledExecutorService` objects. Each of the factory methods has a variant that allows you to explicitly specify a `ThreadFactory`. `newSingleThreadExecutor( )` returns an `ExecutorService` that uses a single thread and an unbounded queue of waiting tasks. `newFixedThreadPool( )` returns an `ExecutorService` that uses a thread pool with the specified number of threads and an unbounded queue. `newCachedThreadPool( )` returns an `ExecutorService` that does not queue tasks but instead creates as many threads as are needed. When a task terminates, its thread is cached for reuse. Cached threads are allowed to terminate if they remain unused for 60 seconds.

`newSingleThreadScheduledExecutor( )` returns a `ScheduledExecutorService` that uses a single thread for running tasks. `newScheduledThreadPool( )` returns a `ScheduledExecutorService` that uses a thread pool of the specified size.

The factory methods of this class typically return instances of `ThreadPoolExecutor` and `ScheduledThreadPoolExecutor`. If the returned objects are cast to these implementing types, they can be configured (to change the thread pool size, for example). If you want to prevent this from happening, use the `unconfigurableExecutorService( )` and `unconfigurableScheduledExecutorService( )` methods to obtain wrapper objects that implement only the `ExecutorService` and `ScheduledExecutorService` methods and do not permit configuration.

Other methods of this class include `callable( )`, which returns a `Callable` object wrapped around a `Runnable` and an optional result, and `defaultThreadFactory( )`, which returns a basic `ThreadFactory` object. Executors also define methods related to access control and the Java security system. A variant of the `callable( )` method wraps a `Callable` around a `java.security.PrivilegedAction`. `privilegedCallable( )` is intended to be invoked from within a `PrivilegedAction` being run with

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`AccessController.doPrivileged( )`. When passed a `Callable` in this way, it returns a new `Callable` that can be used later to invoke the original callable in a privileged access control context, granting it permissions that it would not otherwise have.

```
public class Executors {
// No Constructor
// Public Class Methods
     public static Callable<Object> callable(java.security.PrivilegedAction action);
     public static Callable<Object> callable(Runnable task);
     public static Callable<Object> callable(java.security.PrivilegedExceptionAction action);
     public static <T> Callable<T> callable(Runnable task, T result);
     public static ThreadFactory defaultThreadFactory( );
     public static ExecutorService newCachedThreadPool( );
     public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory);
     public static ExecutorService newFixedThreadPool(int nThreads);
     public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory threadFactory);
     public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize);
     public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize,
        ThreadFactory threadFactory);
     public static ExecutorService newSingleThreadExecutor( );
     public static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory);
     public static ScheduledExecutorService newSingleThreadScheduledExecutor( );
     public static ScheduledExecutorService newSingleThreadScheduledExecutor(ThreadFactory
        threadFactory);
     public static <T> Callable<T> privilegedCallable(Callable<T> callable);
     public static <T> Callable<T> privilegedCallableUsingCurrentClassLoader
        (Callable<T> callable);
     public static ThreadFactory privilegedThreadFactory( );
     public static ExecutorService unconfigurableExecutorService(ExecutorService executor);
     public static ScheduledExecutorService unconfigurableScheduledExecutorService
        (ScheduledExecutorService executor);
}
```

ExecutorService java.util.concurrent

Java 5.0

This interface extends `Executor` to add methods to obtain a `Future` result of the asynchronous execution of a `Callable` task. It also adds methods for graceful termination or shutdown of an `ExecutorService`. `ThreadPoolExecutor` is a useful and highly configurable implementation of this interface. An easy way to obtain instances of this class is through the factory methods of the `Executors` utility class. Note that `ExecutorService` is not a generic type; it does not declare any type variables. It does have a number of generic methods, however, that use the type variable *`T`* to represent the result type of `Callable` and `Future` objects.

The `submit( )` method allows you to submit a `Callable<T>` object to an `ExecutorService` for execution. Typical `ExecutorService` implementations invoke the `call( )` method of the `Callable` on another thread, and the return value (of type *`T`*) of the method is therefore not available when the call to `submit( )` returns.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`submit( )` therefore returns a `Future<T>` object: the promise of a return value of type *T* at some point in the future. See the `Future` interface for further details.

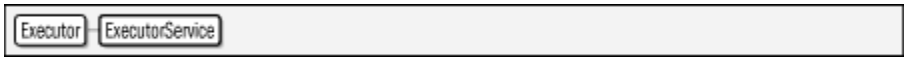
Two variants on the `submit( )` method accept a `java.lang.Runnable` task instead of a `Callable` task. The `run( )` method of a `Runnable` has no return value, so the two-argument version of `submit( )` accepts a dummy return value of type *T* and returns a `Future<T>` that makes this dummy value available when the `Runnable` has completed running. The other `Runnable` variant of the `submit( )` method takes no return value and returns a `Future<?>` value. The `get( )` method of this `Future` object returns `null` when the `Runnable` is done.

Other `ExecutorService` methods execute `Callable` objects synchronously. `invokeAll( )` is passed a `java.util.Collection` of `Callable<T>` tasks. It executes them and blocks until all have completed, or until an optionally specified timeout has elapsed. `invokeAll( )` returns the results of the tasks as a `List` of `Future<T>` objects. Note that a `Callable<T>` task can complete either by returning a result of type `T` or by throwing an exception.

`invokeAny( )` is also passed a `Collection` of `Callable<T>` objects. It blocks until any one of these `Callable` tasks has returned a value of type *T* and returns that value. Tasks that terminate by throwing an exception are ignored. If all tasks throw an exception, `invokeAny( )` throws an `ExecutionException`. Before `invokeAny( )` returns, it cancels the execution of any still-running `Callable` tasks. Like `invokeAll( )`, `invokeAny( )` has a variant with a timeout value.

`ExecutorService` defines several methods for gracefully shutting down the service. `shutdown( )` puts the `ExecutorService` into a special state in which no new tasks may be submitted for execution, but all currently running tasks continue running. `isShutdown( )` returns `true` if the `ExecutorService` has entered this state. `awaitTermination( )` blocks until all executing tasks in an `ExecutorService` that was shut down are completed (or until a specified timeout elapses). Once this has occurred, the `isTerminated( )` method returns `true`. The `shutdownNow( )` method shuts down an `ExecutorService` more abruptly: it attempts to abort all currently executing tasks (typically via `Thread.interrupt( )`) and returns a `List` of the tasks that have not yet started executing.

Figure 16-84. java.util.concurrent.ExecutorService




```
public interface ExecutorService extends Executor {
// Public Instance Methods
     boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException;
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    <T> java.util.List<Future<T>> invokeAll(java.util.Collection<Callable<T>> tasks)
       throws InterruptedException;
    <T> java.util.List<Future<T>> invokeAll(java.util.Collection<Callable<T>> tasks,
       long timeout, TimeUnit unit) throws InterruptedException;
    <T> T invokeAny(java.util.Collection<Callable<T>> tasks)
       throws InterruptedException, ExecutionException;
    <T> T invokeAny(java.util.Collection<Callable<T>> tasks, long timeout, TimeUnit unit)
       throws InterruptedException, ExecutionException, TimeoutException;
    boolean isShutdown( );
    boolean isTerminated( );
    void shutdown( );
    java.util.List<Runnable> shutdownNow( );
    <T> Future<T> submit(Callable<T> task);
    Future<?> submit(Runnable task);
    <T> Future<T> submit(Runnable task, T result);
}
```

Implementations

`AbstractExecutorService`, `ScheduledExecutorService`

Passed To

`Executors.unconfigurableExecutorService( )`

Returned By

`Executors.{newCachedThreadPool( ), newFixedThreadPool( ), newSingleThreadExecutor( ), unconfigurableExecutorService( )}`

Future<V> java.util.concurrent

Java 5.0

This interface represents the result of a computation that may not be available until some time in the future. `Future` is a generic type, with a type variable *`V`*. *`V`* represents the type of the future value to be returned by the `get( )` method. A `Future<V>` value is typically obtained by submitting a `Callable<V>` to an `ExecutorService` for asynchronous execution.

The key method of the `Future` interface is `get( )`. It returns the result (of type *`V`*) of the computation, blocking, if necessary, until that result is ready. `get( )` throws a `CancellationException` if the computation is canceled with the `cancel( )` method before it completes. If the computation throws an exception of its own (as the `Callable.call( )` method can), `get( )` throws an `ExecutionException` wrapped around that exception. Additionally, the timed version of the `get( )` method throws a `TimeoutException` if the timeout elapses before the computation completes.

As noted above, the computation represented by a `Future` object can be canceled by calling its `cancel( )` method. This method returns `true` if the computation was canceled successfully, and `false` otherwise. If you pass `false` to `cancel( )`, any computation

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

that has started running is allowed to complete. In this case, only computations that have not yet started can be canceled. If you pass `true` to the `cancel( )` method, running computations are interrupted with `Thread.interrupt( )`. Note, however, that interrupting a thread does not guarantee that it will stop running.

`isCancelled( )` returns `true` if a `Future` was canceled before it completed (either by returning a value or throwing an exception). `isDone( )` returns `true` if the computation represented by a `Future` is finished running. This may be because it returned a value, threw an exception, or was canceled. If `isDone( )` returns `true`, the `get( )` method does not block.

```
public interface Future<V> {
// Public Instance Methods
     boolean cancel(boolean mayInterruptIfRunning);
     V get( ) throws InterruptedException, ExecutionException;
     V get(long timeout, TimeUnit unit) throws InterruptedException,
        ExecutionException, TimeoutException;
     boolean isCancelled( );
     boolean isDone( );
}
```

Implementations

`FutureTask`, `ScheduledFuture`

Returned By

Too many methods to list.

FutureTask<V> **java.util.concurrent**

Java 5.0 ***runnable***

This class is a `Runnable` wrapper around a `Callable` object (or around another `Runnable`). `FutureTask` is a generic type and the type variable `V` represents the return type of the wrapped `Callable` object. `AbstractExecutorService` uses `FutureTask` to convert `Callable` objects passed to the `submit( )` method into `Runnable` objects it can pass to the `execute( )` method.

`FutureTask` also implements the `Future` interface, which means that the `get( )` method waits for the `run( )` method to complete and provides access to the result (or exception) of the `Callable`'s execution.

The protected methods `set( )` and `setException( )` are invoked when the `Callable` returns a value or throws an exception. `done( )` is invoked when the `Callable` completes or is canceled. Subclasses can override any of these methods to insert hooks for notification, logging, and so on.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-85. java.util.concurrent.FutureTask<V>

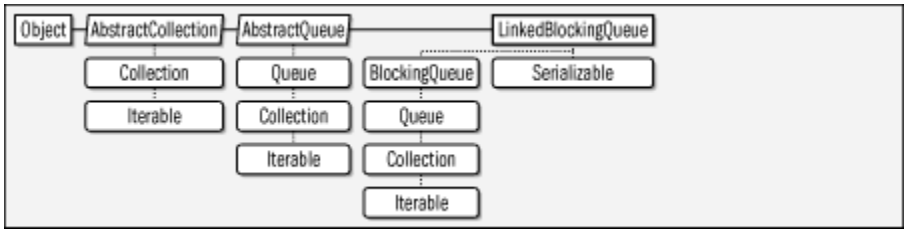

```
public class FutureTask<V> implements Future<V>, Runnable {
// Public Constructors
     public FutureTask(Callable<V> callable);
     public FutureTask(Runnable runnable, V result);
// Methods Implementing Future
     public boolean cancel(boolean mayInterruptIfRunning);
     public V get( ) throws InterruptedException, ExecutionException;
     public V get(long timeout, TimeUnit unit) throws InterruptedException,
        ExecutionException, TimeoutException;
     public boolean isCancelled( );
     public boolean isDone( );
// Methods Implementing Runnable
     public void run( );
// Protected Instance Methods
     protected void done( );                                          empty
     protected boolean runAndReset( );
     protected void set(V v);
     protected void setException(Throwable t);
}
```

LinkedBlockingQueue<E> java.util.concurrent

Java 5.0 ***serializable collection***

This threadsafe class implements the `BlockingQueue` interface based on a linked-list data structure. It orders elements on a first-in, first-out (FIFO) basis. You may specify a maximum queue capacity, creating a bounded queue. The default capacity is `Integer.MAX_VALUE`, which is effectively unbounded. `null` elements are not permitted.

Figure 16-86. java.util.concurrent.LinkedBlockingQueue<E>




```
public class LinkedBlockingQueue<E> extends java.util.AbstractQueue<E>
implements BlockingQueue<E>, Serializable {
// Public Constructors
     public LinkedBlockingQueue( );
     public LinkedBlockingQueue(int capacity);
     public LinkedBlockingQueue(java.util.Collection<? extends E> c);
// Methods Implementing BlockingQueue
     public int drainTo(java.util.Collection<? super E> c);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

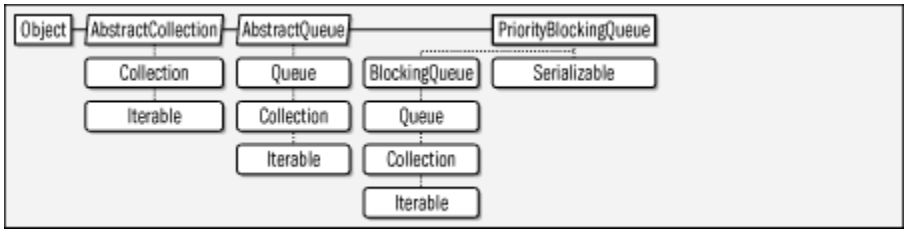
```
     public int drainTo(java.util.Collection<? super E> c, int maxElements);
     public boolean offer(E o);
     public boolean offer(E o, long timeout, TimeUnit unit) throws InterruptedException;
     public E poll(long timeout, TimeUnit unit) throws InterruptedException;
     public void put(E o) throws InterruptedException;
     public int remainingCapacity( );
     public E take( ) throws InterruptedException;
// Methods Implementing Collection
     public void clear( );
     public java.util.Iterator<E> iterator( );
     public boolean remove(Object o);
     public int size( );
     public Object[ ] toArray( );
     public <T> T[ ] toArray(T[ ] a);
// Methods Implementing Queue
     public E peek( );
     public E poll( );
// Public Methods Overriding AbstractCollection
     public String toString( );
}
```

PriorityBlockingQueue<E> java.util.concurrent

Java 5.0 ***serializable collection***

This threadsafe class implements the `BlockingQueue` interface. It is an unbounded queue that orders its elements according to a `Comparator`, or, for `Comparable` elements, according to their `compareTo( )` method. The head of the queue (the next element to be removed) is always the smallest element. Note that the `Iterator` returned by the `iterator( )` method is not guaranteed to return elements in this order. See also `java.util.PriorityQueue`.

Figure 16-87. java.util.concurrent.PriorityBlockingQueue<E>




```
public class PriorityBlockingQueue<E> extends java.util.AbstractQueue<E>
 implements BlockingQueue<E>, Serializable {
// Public Constructors
     public PriorityBlockingQueue( );
     public PriorityBlockingQueue(int initialCapacity);
     public PriorityBlockingQueue(java.util.Collection<? extends E> c);
     public PriorityBlockingQueue(int initialCapacity, java.util.Comparator<? super E>
        comparator);
// Public Instance Methods
     public java.util.Comparator<? super E> comparator( );
// Methods Implementing BlockingQueue
     public boolean add(E o);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public int drainTo(java.util.Collection<? super E> c);
    public int drainTo(java.util.Collection<? super E> c, int maxElements);
    public boolean offer(E o);
    public boolean offer(E o, long timeout, TimeUnit unit);
    public E poll(long timeout, TimeUnit unit) throws InterruptedException;
    public void put(E o);
    public int remainingCapacity( );
    public E take( ) throws InterruptedException;
// Methods Implementing Collection
    public void clear( );
    public boolean contains(Object o);
    public java.util.Iterator<E> iterator( );
    public boolean remove(Object o);
    public int size( );
    public Object[ ] toArray( );
    public <T> T[ ] toArray(T[ ] a);
// Methods Implementing Queue
    public E peek( );
    public E poll( );
// Public Methods Overriding AbstractCollection
    public String toString( );
}
```

RejectedExecutionException java.util.concurrent

Java 5.0 ***serializable unchecked***

An exception of this type is thrown by an `Executor` when it cannot accept a task for execution. When a `ThreadPoolExecutor` cannot accept a task, it attempts to invoke a `RejectedExecutionHandler`. `ThreadPoolExecutor` defines several nested implementations of that handler interface that can handle the rejected task without throwing an exception of this type.

Figure 16-88. java.util.concurrent.RejectedExecutionException


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

RejectedExecutionHandler java.util.concurrent

Java 5.0

This interface defines an API for a handler method invoked by a `ThreadPoolExecutor` when its `execute( )` method cannot accept any more `Runnable` objects. This can occur when both the thread pool and the queue of waiting tasks is full, or when the `ThreadPoolExecutor` has been shut down. Register an instance of this class with the `setRejectedExecutionHandler( )` method of `ThreadPoolExecutor`. `ThreadPoolExecutor` includes several predefined implementations of this interface as static member classes. If the `rejectedExecution( )` method cannot arrange for the `Runnable` to be run and does not wish to simply discard that task, it should throw a `RejectedExecutionException` which propagates up to the caller that submitted the task for execution.

```
public interface RejectedExecutionHandler {
// Public Instance Methods
     void rejectedExecution(Runnable r, ThreadPoolExecutor executor);
}
```

Implementations

`ThreadPoolExecutor.AbortPolicy`,
`ThreadPoolExecutor.CallerRunsPolicy`,
`ThreadPoolExecutor.DiscardOldestPolicy`,
`ThreadPoolExecutor.DiscardPolicy`

Passed To

`ScheduledThreadPoolExecutor.ScheduledThreadPoolExecutor( )`,
`ThreadPoolExecutor.{setRejectedExecutionHandler( )`,
`ThreadPoolExecutor( )}`

Returned By

`ThreadPoolExecutor.getRejectedExecutionHandler( )`

ScheduledExecutorService java.util.concurrent

Java 5.0

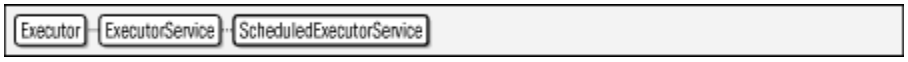
This interface extends `Executor` and `ExecutorService` to add methods for scheduling `Callable` or `Runnable` tasks for future execution on a one-time basis or a repeating basis. The `schedule( )` methods schedule a `Callable` or a `Runnable` task for one-time execution after a specified delay. The delay is specified by a `long` plus a `TimeUnit`. When a `Callable<V>` is scheduled, the result is a `ScheduledFuture<V>`. This is like a

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`Future<V>` object but also implements the `Delay` interface so you can call `getDelay( )` to find out how much time remains before execution begins. If you `schedule( )` a `Runnable` object, the result is a `ScheduledFuture<?>`. Since a `Runnable` has no return value, the `get( )` method of this `ScheduledFuture` returns `null`, but the `cancel( )`, `getDelay( )`, and `isDone( )` methods remain useful.

`ScheduledExecutorService` provides two alternatives for scheduling `Runnable` tasks for repeated execution. (See also `java.util.Timer`, which has similar methods.) `scheduleAtFixedRate( )` begins the first execution of the `Runnable` after *`initialDelay`* time units, and begins subsequent executions at multiples of *`period`* time units after that. This means that the `Runnable` runs at a fixed rate, regardless of how long each execution takes. `scheduleWithFixedDelay( )` also begins the first execution after *`initialDelay`* time units. But it waits for this first execution (and all subsequent executions) to complete before scheduling the next execution for *`delay`* time units in the future. Both methods return a `ScheduledFuture` object that you can use to `cancel( )` the repeated execution of tasks. If the task is not canceled, the `ScheduledExecutorService` runs it repeatedly until the service is shut down (see `ExecutorService`) or the `Runnable` throws an exception.

Figure 16-89. java.util.concurrent.ScheduledExecutorService




```
public interface ScheduledExecutorService extends ExecutorService {
// Public Instance Methods
     <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit);
     ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit);
     ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay,
        long period, TimeUnit unit);
     ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay,
        long delay, TimeUnit unit);
}
```

Implementations

`ScheduledThreadPoolExecutor`

Passed To

`Executors.unconfigurableScheduledExecutorService( )`

Returned By

`Executors.{newScheduledThreadPool( ), newSingleThreadScheduledExecutor( ), unconfigurableScheduledExecutorService( )}`

ScheduledFuture<V> **java.util.concurrent**

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 5.0 ***comparable***

This interface extends `Future` and `Delayed` and adds no methods of its own. A `ScheduledFuture` represents a computation and the future result of that computation just as `Future` does, but it adds a `getDelay( )` method that returns the amount of time until the computation begins. See `ScheduledExecutorService`.

Figure 16-90. java.util.concurrent.ScheduledFuture<V>


```
public interface ScheduledFuture<V> extends DelayedFuture<V> {
}
```

Returned By

`ScheduledExecutorService.{schedule( ),scheduleAtFixedRate( ), scheduleWithFixedDelay( )},ScheduledThreadPoolExecutor. {schedule( ),scheduleAtFixedRate( ),scheduleWithFixedDelay( )}`

ScheduledThreadPoolExecutor **java.util.concurrent**

Java 5.0

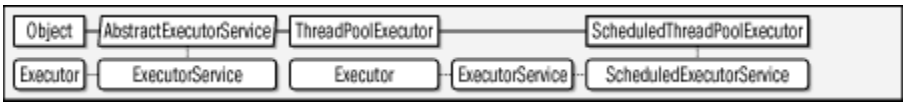
This class extends `ThreadPoolExecutor` to implement the methods of the `ScheduledExecutorService` interface to allow tasks to be submitted for execution once or repeatedly at some scheduled time in the future. Instances of this class are usually obtained through the static factory methods of the `Executors` utility class. You can also explicitly create one with the `ScheduledThreadPoolExecutors( )` constructor. `ScheduledThreadPoolExecutor` always creates its own unbounded work queue, which means that you cannot pass a queue to the constructor. Also, there is no need to specify a *`maximumPoolSize`* since this configuration parameter is irrelevant with unbounded queues.

Note that tasks submitted to a `ScheduledThreadPoolExecutor` are not guaranteed to run at the scheduled time. That is the time at which they first become eligible to run. If all threads are busy with other tasks, however, eligible tasks may get queued up to run later.

This class provides functionality similar to `java.util.Timer` but adds multithreaded capability and the ability to work with `Callable` and `Future` objects.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-91. java.util.concurrent.ScheduledThreadPoolExecutor




```
public class ScheduledThreadPoolExecutor extends ThreadPoolExecutor
        implements ScheduledExecutorService {
// Public Constructors
     public ScheduledThreadPoolExecutor(int corePoolSize);
     public ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory);
     public ScheduledThreadPoolExecutor(int corePoolSize, RejectedExecutionHandler handler);
     public ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory,
 RejectedExecutionHandler handler);
// Public Instance Methods
     public boolean getContinueExistingPeriodicTasksAfterShutdownPolicy( );
     public boolean getExecuteExistingDelayedTasksAfterShutdownPolicy( );
     public void setContinueExistingPeriodicTasksAfterShutdownPolicy(boolean value);
     public void setExecuteExistingDelayedTasksAfterShutdownPolicy(boolean value);
// Methods Implementing Executor
     public void execute(Runnable command);
// Methods Implementing ExecutorService
     public void shutdown( );
     public java.util.List<Runnable> shutdownNow( );
     public Future<?> submit(Runnable task);
     public <T> Future<T> submit(Callable<T> task);
     public <T> Future<T> submit(Runnable task, T result);
// Methods Implementing ScheduledExecutorService
     public <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit);
     public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit);
     public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay,
        long period, TimeUnit unit);
     public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay,
        long delay, TimeUnit unit);
// Public Methods Overriding ThreadPoolExecutor
     public BlockingQueue<Runnable> getQueue( );
     public boolean remove(Runnable task);
}
```

Semaphore java.util.concurrent

Java 5.0 ***serializable***

This class implements *semaphores*, a classic thread synchronization primitive that can be used to implement mutual exclusion and wait/notify-style thread synchronization. A `Semaphore` maintains some fixed number (specified when the `Semaphore( )` constructor is called) of *`permits`*. The `acquire( )` method blocks until a permit is available, then decrements the number of available permits and returns. The `release( )` method does the reverse: it increments the number of permits, possibly unblocking a thread waiting in `acquire( )`.

If you pass `true` as the second argument to the `Semaphore( )` constructor, the semaphore treats waiting threads fairly by placing them on a FIFO queue in the order they

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

called `acquire( )` and granting permits to the threads in this order. This prevents thread starvation.

Figure 16-92. java.util.concurrent.Semaphore


```
public class Semaphore implements Serializable {
// Public Constructors
     public Semaphore(int permits);
     public Semaphore(int permits, boolean fair);
// Public Instance Methods
     public void acquire( ) throws InterruptedException;
     public void acquire(int permits) throws InterruptedException;
     public void acquireUninterruptibly( );
     public void acquireUninterruptibly(int permits);
     public int availablePermits( );
     public int drainPermits( );
     public final int getQueueLength( );
     public final boolean hasQueuedThreads( );
     public boolean isFair( );
     public void release( );
     public void release(int permits);
     public boolean tryAcquire( );
     public boolean tryAcquire(int permits);
     public boolean tryAcquire(long timeout, TimeUnit unit)
        throws InterruptedException;
     public boolean tryAcquire(int permits, long timeout, TimeUnit unit)
        throws InterruptedException;
// Public Methods Overriding Object
     public String toString( );
// Protected Instance Methods
     protected java.util.Collection<Thread> getQueuedThreads( );
     protected void reducePermits(int reduction);
}
```

SynchronousQueue<E> **java.util.concurrent**

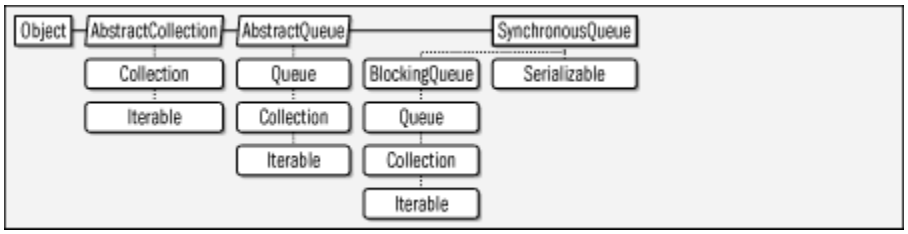
Java 5.0 ***serializable collection***

This `BlockingQueue` implementation is the degenerate case of a bounded queue with a capacity of zero. Every call to `put( )` blocks until a corresponding call to `take( )`, and vice versa. You can think of this as an `Exchanger` that does only a one-way exchange.

The `size( )` and `remainingCapacity( )` methods always return 0. The `peek( )` method always returns `null`. The `iterator( )` method returns an `Iterator` for which the `hasNext( )` method returns `false`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-93. java.util.concurrent.SynchronousQueue<E>




```
public class SynchronousQueue<E> extends java.util.AbstractQueue<E>
        implements BlockingQueue<E>, Serializable {
// Public Constructors
     public SynchronousQueue( );
     public SynchronousQueue(boolean fair);
// Methods Implementing BlockingQueue
     public int drainTo(java.util.Collection<? super E> c);
     public int drainTo(java.util.Collection<? super E> c, int maxElements);
     public boolean offer(E o);
     public boolean offer(E o, long timeout, TimeUnit unit) throws InterruptedException;
     public E poll(long timeout, TimeUnit unit) throws InterruptedException;
     public void put(E o) throws InterruptedException;
     public int remainingCapacity( );                                    constant
     public E take( ) throws InterruptedException;
// Methods Implementing Collection
     public void clear( );                                               empty
     public boolean contains(Object o);                                 constant
     public boolean containsAll(java.util.Collection<?> c);
     public boolean isEmpty( );                                          constant default:true
     public java.util.Iterator<E> iterator( );
     public boolean remove(Object o);                                   constant
     public boolean removeAll(java.util.Collection<?> c);       constant
     public boolean retainAll(java.util.Collection<?> c);       constant
     public int size( );                                                 constant
     public Object[ ] toArray( );
     public <T> T[ ] toArray(T[ ] a);
// Methods Implementing Queue
     public E peek( );                                                   constant
     public E poll( );
}
```

ThreadFactory java.util.concurrent

Java 5.0

An instance of this interface is an object that creates `Thread` objects to run `Runnable` objects. You might define a `ThreadFactory` if you want to set the priority, name, or `ThreadGroup` of the threads used by a `ThreadPoolExecutor`, for example. A number of the factory methods of the `Executors` utility class rely on `ThreadPoolExecutor` and accept a `ThreadFactory` argument.

```
public interface ThreadFactory {
// Public Instance Methods
     Thread newThread(Runnable r);
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Passed To

`Executors.{newCachedThreadPool( ),newFixedThreadPool( ), newScheduledThreadPool( ),newSingleThreadExecutor( ), newSingleThreadScheduledExecutor( )}, ScheduledThreadPoolExecutor.ScheduledThreadPoolExecutor( ), ThreadPoolExecutor.{setThreadFactory( ),ThreadPoolExecutor( )}`

Returned By

`Executors.{defaultThreadFactory( ),privilegedThreadFactory( )}, ThreadPoolExecutor.getThreadFactory( )`

ThreadPoolExecutor java.util.concurrent

Java 5.0

This class implements the `ExecutorService` interface to execute tasks using a highly configurable thread pool. The easiest way to instantiate this class is through the static factory methods of the `Executors` class. If you want a more highly configured thread pool, you can instantiate it directly.

Four configuration parameters must be passed to every `ThreadPoolExecutor( )` constructor; two others are optional. Many of these parameters may also be queried and adjusted after the executor has been created through various `ThreadPoolExecutor` accessor methods. The most important configuration parameters specify the size of the thread pool, and the queue that the executor uses to hold tasks that it cannot currently run. *`corePoolSize`* is the number of threads that the pool should hold under normal usage. As tasks are submitted to the `ThreadPoolExecutor`, a new thread is created for each task until the total number of threads reaches this size.

If *`corePoolSize`* threads have already been created, newly submitted tasks are placed on the work queue. As these core threads finish the tasks they are executing, they `take( )` a new task from the work queue. You must specify the *`workQueue`* when you call the `ThreadPoolExecutor( )` constructor. It may be any `BlockingQueue` object and the behavior of the thread pool depends strongly on the behavior of the queue you specify. Options include an unbounded `LinkedBlockingQueue`, a bounded `ArrayBlockingQueue` with a capacity of your choosing, or even a `SynchronousQueue` which has a capacity of zero and cannot actually accept a task unless a thread is already waiting to execute it.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

If the work queue becomes empty, it is inefficient to leave all the core threads sitting idly waiting for work. Threads are terminated if they are idle for more than the "keep alive" time. You specify this time with the *keepAliveTime* parameter and a `TimeUnit` constant.

If the work queue fills up, the *maximumPoolSize* parameter comes into play. `ThreadPoolExecutor` prefers to maintain *corePoolSize* threads but allows this number to grow up to *maximumPoolSize*. A new thread is created only when the *workQueue* is full. If you specify an unbounded work queue, *maximumPoolSize* is irrelevant because the queue never fills up. If on the other hand you specify a `SynchronousQueue` (which is always full), if none of the existing threads are waiting for a new task, a new thread is always created (up to the `maximumPoolSize` limit).

If a `ThreadPoolExecutor` has already created the maximum number of threads and its work queue is full, it must reject any newly submitted tasks. The default behavior is to throw a `RejectedExecutionException`. You can alter this behavior by specifying a `RejectedExecutionHandler` object to the `ThreadPoolExecutor( )` constructor or with the `setRejectedExecutionHandler( )` method. The four inner classes of this class are implementations of four handlers that address this case. See their individual entries for details.

The final way that you can customize a `ThreadPoolExecutor` is to pass `ThreadFactory` to the constructor or to the `setThreadFactory( )` method. If you do not specify a factory, the `ThreadPoolExecutor` obtains one with `Executors.defaultThreadFactory( )`.

Figure 16-94. java.util.concurrent.ThreadPoolExecutor


```
public class ThreadPoolExecutor extends AbstractExecutorService {
// Public Constructors
     public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
        long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue);
     public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
        long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,
        ThreadFactory threadFactory);
     public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
        long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,
        RejectedExecutionHandler handler);
     public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
        long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,
        ThreadFactory threadFactory,
RejectedExecutionHandler handler);
// Nested Types
     public static class AbortPolicy implements RejectedExecutionHandler;
     public static class CallerRunsPolicy implements RejectedExecutionHandler;
     public static class DiscardOldestPolicy implements RejectedExecutionHandler;
     public static class DiscardPolicy implements RejectedExecutionHandler;
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
// Public Instance Methods
     public int getActiveCount( );
     public long getCompletedTaskCount( );
     public int getCorePoolSize( );
     public long getKeepAliveTime(TimeUnit unit);
     public int getLargestPoolSize( );
     public int getMaximumPoolSize( );
     public int getPoolSize( );
     public BlockingQueue<Runnable> getQueue( );
     public RejectedExecutionHandler getRejectedExecutionHandler( );
     public long getTaskCount( );
     public ThreadFactory getThreadFactory( );
     public boolean isTerminating( );
     public int prestartAllCoreThreads( );
     public boolean prestartCoreThread( );
     public void purge( );
     public boolean remove(Runnable task);
     public void setCorePoolSize(int corePoolSize);
     public void setKeepAliveTime(long time, TimeUnit unit);
     public void setMaximumPoolSize(int maximumPoolSize);
     public void setRejectedExecutionHandler(RejectedExecutionHandler handler);
     public void setThreadFactory(ThreadFactory threadFactory);
// Methods Implementing Executor
     public void execute(Runnable command);
// Methods Implementing ExecutorService
     public boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException;
     public boolean isShutdown( );
     public boolean isTerminated( );
     public void shutdown( );
     public java.util.List<Runnable> shutdownNow( );
// Protected Methods Overriding Object
     protected void finalize( );
// Protected Instance Methods
     protected void afterExecute(Runnable r, Throwable t);              empty
     protected void beforeExecute(Thread t, Runnable r);                empty
     protected void terminated( );                                        empty
}
```

Subclasses

`ScheduledThreadPoolExecutor`

Passed To

`RejectedExecutionHandler.rejectedExecution( )`,
`ThreadPoolExecutor.AbortPolicy.rejectedExecution( )`,
`ThreadPoolExecutor.CallerRunsPolicy.rejectedExecution( )`,
`ThreadPoolExecutor.DiscardOldestPolicy.rejectedExecution( )`,
`ThreadPoolExecutor.DiscardPolicy.rejectedExecution( )`

ThreadPoolExecutor.AbortPolicy java.util.concurrent

Java 5.0

This `RejectedExecutionHandler` implementation simply throws a `RejectedExecutionException`.

```
public static class ThreadPoolExecutor.AbortPolicy implements RejectedExecutionHandler {
// Public Constructors
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public AbortPolicy( );
// Methods Implementing RejectedExecutionHandler
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e);
}
```

ThreadPoolExecutor.CallerRunsPolicy java.util.concurrent

Java 5.0

This `RejectedExecutionHandler` implementation runs the rejected `Runnable` object directly in the calling thread, causing that thread to block until the `Runnable` completes. If the `ThreadPoolExecutor` has been shut down, the `Runnable` is simply discarded instead of being run.

```
public static class ThreadPoolExecutor.CallerRunsPolicy implements RejectedExecutionHandler {
// Public Constructors
    public CallerRunsPolicy( );
// Methods Implementing RejectedExecutionHandler
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e);
}
```

ThreadPoolExecutor.DiscardOldestPolicy java.util.concurrent

Java 5.0

This `RejectedExecutionHandler` implementation discards the rejected `Runnable` if the `ThreadPoolExecutor` has been shut down. Otherwise, it discards the oldest pending task that has not run and tries again to `execute( )` the rejected task.

```
public static class ThreadPoolExecutor.DiscardOldestPolicy implements RejectedExecutionHandler {
// Public Constructors
    public DiscardOldestPolicy( );
// Methods Implementing RejectedExecutionHandler
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e);
}
```

ThreadPoolExecutor.DiscardPolicy java.util.concurrent

Java 5.0

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

This `RejectedExecutionHandler` implementation silently discards the rejected `Runnable`.

```
public static class ThreadPoolExecutor.DiscardPolicy implements RejectedExecutionHandler {
// Public Constructors
     public DiscardPolicy( );
// Methods Implementing RejectedExecutionHandler
     public void rejectedExecution(Runnable r, ThreadPoolExecutor e);     empty
}
```

TimeoutException java.util.concurrent

Java 5.0 ***serializable checked***

An exception of this type is thrown by timed methods to indicate that the specified timeout has elapsed. Other timed methods are able to indicate their timeout status in a `boolean` or other return value.

Figure 16-95. java.util.concurrent.TimeoutException


```
public class TimeoutException extends Exception {
// Public Constructors
     public TimeoutException( );
     public TimeoutException(String message);
}
```

Thrown By

`AbstractExecutorService.invokeAny( )`, `CyclicBarrier.await( )`, `Exchanger.exchange( )`, `ExecutorService.invokeAny( )`, `Future.get( )`, `FutureTask.get( )`

TimeUnit java.util.concurrent

Java 5.0 ***serializable comparable enum***

The constants defined by this enumerated type represent granularities of time. Timeout and delay specifications throughout the `java.util.concurrent` package are specified by a `long` value and `TimeUnit` constant that specifies the interpretation of that value.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`TimeUnit` defines conversion methods that convert values expressed in one unit to values in another unit. More interestingly, it defines convenient alternatives to `Thread.sleep( )`, `Thread.join( )`, and `Object.wait( )`.

Figure 16-96. java.util.concurrent.TimeUnit


```
public enum TimeUnit {
// Enumerated Constants
     NANOSECONDS,
     MICROSECONDS,
     MILLISECONDS,
     SECONDS;
// Public Class Methods
     public static TimeUnit valueOf(String name);
     public static final TimeUnit[ ] values( );
// Public Instance Methods
     public long convert(long duration, TimeUnit unit);
     public void sleep(long timeout) throws InterruptedException;
     public void timedJoin(Thread thread, long timeout) throws InterruptedException;
     public void timedWait(Object obj, long timeout) throws InterruptedException;
     public long toMicros(long duration);
     public long toMillis(long duration);
     public long toNanos(long duration);
     public long toSeconds(long duration);
}
```

Passed To

Too many methods to list.

Package java.util.concurrent.atomic

Java 5.0

This package includes classes that provide atomic operations on boolean, integer, and reference values. Instances of the classes defined here have the properties of `volatile` fields but also add atomic operations like the canonical `compareAndSet( )`, which verifies that the field holds an expected value, and, if it does, sets it to a new value. The classes also define a `weakCompareAndSet( )` method that may be more efficient than `compareAndSet( )` but may also fail to set the value even when the field holds the expected value.

The "Array" classes provide atomic access to arrays of values and provide `volatile` access semantics for array elements, which is not possible with the `volatile` modifier itself. The "FieldUpdater" classes use reflection to provide atomic operations on a named `volatile` field of an existing class. The `AtomicMarkableReference` class and

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`AtomicStampedReference` class maintain a reference value and an associated `boolean` or `int` value and allow the two values to be atomically manipulated together. These classes can be useful in concurrent algorithms that detect concurrent updates with version numbering, for example.

Most implementations of this package rely on low-level atomic instructions in the underlying CPU and perform atomic operations without the overhead of locking.

Classes

```
public class AtomicBoolean implements Serializable;
public class AtomicInteger extends Number implements Serializable;
public class AtomicIntegerArray implements Serializable;
public abstract class AtomicIntegerFieldUpdater<T>;
public class AtomicLong extends Number implements Serializable;
public class AtomicLongArray implements Serializable;
public abstract class AtomicLongFieldUpdater<T>;
public class AtomicMarkableReference<V>;
public class AtomicReference<V> implements Serializable;
public class AtomicReferenceArray<E> implements Serializable;
public abstract class AtomicReferenceFieldUpdater<T, V>;
public class AtomicStampedReference<V>;
```

AtomicBoolean java.util.concurrent.atomic

Java 5.0 ***serializable***

This threadsafe class holds a boolean value. In addition to the `get( )` and `set( )` iterators, it provides atomic `compareAndSet( )`, `weakCompareAndSet( )`, and `getAndSet( )` operations.

Figure 16-97. java.util.concurrent.atomic.AtomicBoolean


```
public class AtomicBoolean implements Serializable {
// Public Constructors
     public AtomicBoolean( );
     public AtomicBoolean(boolean initialValue);
// Public Instance Methods
     public final boolean compareAndSet(boolean expect, boolean update);
     public final boolean get( );
     public final boolean getAndSet(boolean newValue);
     public final void set(boolean newValue);
     public boolean weakCompareAndSet(boolean expect, boolean update);
// Public Methods Overriding Object
     public String toString( );
}
```

AtomicInteger java.util.concurrent.atomic

Java 5.0 ***serializable***

This threadsafe class holds an `int` value. It extends `java.lang.Number`, but unlike the `Integer` class, it is mutable. Access the `int` value with the `get( )` method and the various methods inherited from `Number`. You can set the value with the `set( )` method or through various atomic methods. In addition to the basic `compareAndSet( )` and `weakCompareAndSet( )` methods, this class defines methods for atomic pre-increment, post-increment, pre-decrement and post-decrement operations as well as generalized `addAndGet( )` and `getAndAdd( )` methods. `addAndGet( )` atomically adds the specified amount to the stored value and returns the new value. `getAndAdd( )` atomically returns the current value and then adds the specified amount to it.

Figure 16-98. java.util.concurrent.atomic.AtomicInteger


```
public class AtomicInteger extends Number implements Serializable {
// Public Constructors
     public AtomicInteger( );
     public AtomicInteger(int initialValue);
// Public Instance Methods
     public final int addAndGet(int delta);
     public final boolean compareAndSet(int expect, int update);
     public final int decrementAndGet( );
     public final int get( );
     public final int getAndAdd(int delta);
     public final int getAndDecrement( );                                  default:0
     public final int getAndIncrement( );                                  default:-1
     public final int getAndSet(int newValue);
     public final int incrementAndGet( );
     public final void set(int newValue);
     public final boolean weakCompareAndSet(int expect, int update);
// Public Methods Overriding Number
     public double doubleValue( );
     public float floatValue( );
     public int intValue( );
     public long longValue( );
// Public Methods Overriding Object
     public String toString( );
}
```

AtomicIntegerArray java.util.concurrent.atomic

Java 5.0 ***serializable***

This class holds an array of `int` values. It provides threadsafe access to the array elements, treating each as if it was a `volatile` field, and defines atomic operations on them. The methods of this class are like those of `AtomicInteger`, except that each has an additional parameter that specifies the array index. Create an `AtomicIntegerArray` by specifying the desired array length or an actual `int[ ]` from which initial values can be copied.

Figure 16-99. java.util.concurrent.atomic.AtomicIntegerArray


```
public class AtomicIntegerArray implements Serializable {
// Public Constructors
     public AtomicIntegerArray(int[ ] array);
     public AtomicIntegerArray(int length);
// Public Instance Methods
     public final int addAndGet(int i, int delta);
     public final boolean compareAndSet(int i, int expect, int update);
     public final int decrementAndGet(int i);
     public final int get(int i);
     public final int getAndAdd(int i, int delta);
     public final int getAndDecrement(int i);
     public final int getAndIncrement(int i);
     public final int getAndSet(int i, int newValue);
     public final int incrementAndGet(int i);
     public final int length( );
     public final void set(int i, int newValue);
     public final boolean weakCompareAndSet(int i, int expect, int update);
// Public Methods Overriding Object
     public String toString( );
}
```

AtomicIntegerFieldUpdater<T> java.util.concurrent.atomic

Java 5.0

This class uses `java.lang.reflect` to provide atomic operations for named volatile `int` fields within existing types. Obtain an instance of this class with the `newUpdater( )` factory method. Pass the name of the field (which must have been declared `volatile int`) to be updated and the class that it is defined within to this factory method. The instance methods of the resulting `AtomicIntegerFieldUpdater` object are like those of the `AtomicInteger` class but require you to specify the object whose field is to be manipulated. This is a generic type, and the type variable *`T`* represents the type whose `volatile int` field is being updated.

```
public abstract class AtomicIntegerFieldUpdater<T> {
// Protected Constructors
     protected AtomicIntegerFieldUpdater( );
// Public Class Methods
     public static <U> AtomicIntegerFieldUpdater<U> newUpdater(Class<U> tclass,
        String fieldName);
// Public Instance Methods
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public int addAndGet(T obj, int delta);
    public abstract boolean compareAndSet(T obj, int expect, int update);
    public int decrementAndGet(T obj);
    public abstract int get(T obj);
    public int getAndAdd(T obj, int delta);
    public int getAndDecrement(T obj);
    public int getAndIncrement(T obj);
    public int getAndSet(T obj, int newValue);
    public int incrementAndGet(T obj);
    public abstract void set(T obj, int newValue);
    public abstract boolean weakCompareAndSet(T obj, int expect, int update);
}
```

AtomicLong java.util.concurrent.atomic

Java 5.0 ***serializable***

This threadsafe class holds a mutable `long` value and defines atomic operations on that value. It behaves just like `AtomicInteger`, with the substitution of `long` for `int`.

Figure 16-100. java.util.concurrent.atomic.AtomicLong


```
public class AtomicLong extends Number implements Serializable {
// Public Constructors
    public AtomicLong( );
    public AtomicLong(long initialValue);
// Public Instance Methods
    public final long addAndGet(long delta);
    public final boolean compareAndSet(long expect, long update);
    public final long decrementAndGet( );
    public final long get( );
    public final long getAndAdd(long delta);
    public final long getAndDecrement( );                          default:0
    public final long getAndIncrement( );                          default:-1
    public final long getAndSet(long newValue);
    public final long incrementAndGet( );
    public final void set(long newValue);
    public final boolean weakCompareAndSet(long expect, long update);
// Public Methods Overriding Number
    public double doubleValue( );
    public float floatValue( );
    public int intValue( );
    public long longValue( );
// Public Methods Overriding Object
    public String toString( );
}
```

AtomicLongArray java.util.concurrent.atomic

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 5.0 ***serializable***

This threadsafe class provides atomic operations for an array of `long` values. See `AtomicIntegerArray`, which offers the equivalent operations for `int` arrays.

Figure 16-101. java.util.concurrent.atomic.AtomicLongArray


```
public class AtomicLongArray implements Serializable {
// Public Constructors
     public AtomicLongArray(long[ ] array);
     public AtomicLongArray(int length);
// Public Instance Methods
     public long addAndGet(int i, long delta);
     public final boolean compareAndSet(int i, long expect, long update);
     public final long decrementAndGet(int i);
     public final long get(int i);
     public final long getAndAdd(int i, long delta);
     public final long getAndDecrement(int i);
     public final long getAndIncrement(int i);
     public final long getAndSet(int i, long newValue);
     public final long incrementAndGet(int i);
     public final int length( );
     public final void set(int i, long newValue);
     public final boolean weakCompareAndSet(int i, long expect, long update);
// Public Methods Overriding Object
     public String toString( );
}
```

AtomicLongFieldUpdater<T> **java.util.concurrent.atomic**

Java 5.0

This class uses `java.lang.reflect` to define atomic operations for named `volatile long` fields of a specified class. See `AtomicIntegerFieldUpdater`, which is very similar.

```
public abstract class AtomicLongFieldUpdater<T> {
// Protected Constructors
     protected AtomicLongFieldUpdater( );
// Public Class Methods
     public static <U> AtomicLongFieldUpdater<U> newUpdater(Class<U> tclass, String fieldName);
// Public Instance Methods
     public long addAndGet(T obj, long delta);
     public abstract boolean compareAndSet(T obj, long expect, long update);
     public long decrementAndGet(T obj);
     public abstract long get(T obj);
     public long getAndAdd(T obj, long delta);
     public long getAndDecrement(T obj);
     public long getAndIncrement(T obj);
     public long getAndSet(T obj, long newValue);
     public long incrementAndGet(T obj);
     public abstract void set(T obj, long newValue);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
     public abstract boolean weakCompareAndSet(T obj, long expect, long update);
}
```

AtomicMarkableReference<V> java.util.concurrent.atomic

Java 5.0

This threadsafe class holds a mutable reference to an object of type `V` and also holds a mutable `boolean` value or "mark." It defines atomic operations and volatile access semantics for the reference and the mark. The `set( )` method unconditionally sets the reference and mark value. The `get( )` method queries both, returning the reference as its return value, and storing the current value of the mark in element 0 of the specified `boolean` array. The reference and mark can also be queried individually (and nonatomically) with `getReference( )` and `isMarked( )`.

The atomic `compareAndSet( )` and `weakCompareAndSet( )` methods take expected and new values for both the reference and the mark, and neither is set to its new value unless both match their expected values. `attemptMark( )` atomically sets the value of the mark but only if the reference is equal to the expected value. Like `weakCompareAndSet( )`, this method may fail spuriously, even if the reference does equal the expected value. Repeated invocation eventually succeeds, however, as long as the expected value is correct, and other threads are not continuously changing the reference value.

```
public class AtomicMarkableReference<V> {
// Public Constructors
     public AtomicMarkableReference(V initialRef, boolean initialMark);
// Public Instance Methods
     public boolean attemptMark(V expectedReference, boolean newMark);
     public boolean compareAndSet(V expectedReference, V newReference,
        boolean expectedMark, boolean newMark);
     public V get(boolean[ ] markHolder);
     public V getReference( );
     public boolean isMarked( );
     public void set(V newReference, boolean newMark);
     public boolean weakCompareAndSet(V expectedReference, V newReference,
        boolean expectedMark, boolean newMark);
}
```

AtomicReference<V> java.util.concurrent.atomic

Java 5.0 ***serializable***

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

This threadsafe class holds a mutable reference to an object of type `V`, provides `volatile` access semantics, and defines atomic operations for manipulating that value. `get( )` and `set( )` are ordinary accessor methods for the reference. `compareAndSet( )`, `weakCompareAndSet( )`, and `getAndSet( )` perform the two named operations atomically. `compareAndSet( )` is the canonical atomic operation: the reference is compared to an expected value, and, if it matches, is set to a new value. `compareAndSet( )` returns true if it set the value or `false` otherwise. `weakCompareAndSet( )` is similar but may fail to set the reference even if it does match the expected value (it is guaranteed to succeed eventually if the operation is repeatedly retried, however).

Figure 16-102. java.util.concurrent.atomic.AtomicReference<V>


```
public class AtomicReference<V> implements Serializable {
// Public Constructors
     public AtomicReference( );
     public AtomicReference(V initialValue);
// Public Instance Methods
     public final boolean compareAndSet(V expect, V update);
     public final V get( );
     public final V getAndSet(V newValue);
     public final void set(V newValue);
     public final boolean weakCompareAndSet(V expect, V update);
// Public Methods Overriding Object
     public String toString( );
}
```

AtomicReferenceArray<E> java.util.concurrent.atomic

Java 5.0 ***serializable***

This threadsafe class holds an array of elements of type `E`. It provides `volatile` access semantics for these array elements and defines atomic operations for manipulating them. Its methods are like those of `AtomicReference` with the addition of a parameter that specifies the array index of the desired element.

Figure 16-103. java.util.concurrent.atomic.AtomicReferenceArray<E>


```
public class AtomicReferenceArray<E> implements Serializable {
// Public Constructors
     public AtomicReferenceArray(E[ ] array);
     public AtomicReferenceArray(int length);
// Public Instance Methods
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public final boolean compareAndSet(int i, E expect, E update);
    public final E get(int i);
    public final E getAndSet(int i, E newValue);
    public final int length( );
    public final void set(int i, E newValue);
    public final boolean weakCompareAndSet(int i, E expect, E update);
// Public Methods Overriding Object
    public String toString( );
}
```

AtomicReferenceFieldUpdater<T,V> java.util.concurrent.atomic

Java 5.0

This threadsafe class uses `java.lang.reflect` to provide atomic operations for a named volatile field of type *V* within an object of type *T*. Its instance methods are like those of `AtomicReference` and the static `newUpdater( )` factory method is like that of `AtomicIntegerFieldUpdater`.

```
public abstract class AtomicReferenceFieldUpdater<T,V> {
// Protected Constructors
    protected AtomicReferenceFieldUpdater( );
// Public Class Methods
    public static <U,W> AtomicReferenceFieldUpdater<U,W> newUpdater(Class<U> tclass,
       Class<W> vclass, String fieldName);
// Public Instance Methods
    public abstract boolean compareAndSet(T obj, V expect, V update);
    public abstract V get(T obj);
    public V getAndSet(T obj, V newValue);
    public abstract void set(T obj, V newValue);
    public abstract boolean weakCompareAndSet(T obj, V expect, V update);
}
```

AtomicStampedReference<V> java.util.concurrent.atomic

Java 5.0

This threadsafe class holds a mutable reference to an object of type *V* and also holds a mutable `int` value or "stamp." It defines atomic operations and volatile access semantics for the reference and the stamp. This class works just like `AtomicMarkableReference` except that an `int` "stamp" replaces the `boolean` "mark." See `AtomicMarkableReference` for further details.

```
public class AtomicStampedReference<V> {
// Public Constructors
    public AtomicStampedReference(V initialRef, int initialStamp);
// Public Instance Methods
    public boolean attemptStamp(V expectedReference, int newStamp);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public boolean compareAndSet(V expectedReference, V newReference,
       int expectedStamp, int newStamp);
    public V get(int[ ] stampHolder);
    public V getReference( );
    public int getStamp( );
    public void set(V newReference, int newStamp);
    public boolean weakCompareAndSet(V expectedReference, V newReference,
       int expectedStamp, int newStamp);
}
```

Package java.util.concurrent.locks

Java 5.0

This package defines `Lock` and associated `Condition` interfaces as well as concrete implementations (such as `ReentrantLock`) that provide an alternative to locking with `synchronized` blocks and methods and to waiting with the `wait( )`, `notify( )`, and `notifyAll( )` methods of `Object`.

Although `Lock` and `Condition` are somewhat more complex to use than the built-in locking, waiting, and notification mechanisms of `Object`, they are also more flexible. `Lock`, for example, does not require that locks be block-structured and enables algorithms such as "hand-over-hand locking" for traversing linked data structures. A thread waiting to acquire a `Lock` can time out or be interrupted, which is not possible with `synchronized` locking. Also, more than one `Condition` can be associated with a given `Lock`, which is simply not possible with `Object`-based locking and waiting.

The `ReadWriteLock` interface and its `ReentrantReadWriteLock` implementation allow multiple concurrent readers but only a single writer thread to hold the lock.

Interfaces

```
public interface Condition;
public interface Lock;
public interface ReadWriteLock;
```

Classes

```
public abstract class AbstractQueuedSynchronizer implements Serializable;
public class AbstractQueuedSynchronizer.ConditionObject implements Condition, Serializable;
public class LockSupport;
public class ReentrantLock implements Lock, Serializable;
public class ReentrantReadWriteLock implements ReadWriteLock, Serializable;
public static class ReentrantReadWriteLock.ReadLock implements Lock, Serializable;
public static class ReentrantReadWriteLock.WriteLock implements Lock, Serializable;
```

AbstractQueuedSynchronizer java.util.concurrent.locks

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 5.0 ***serializable***

This abstract class is a low-level utility. A concrete subclass can be used as a helper class for implementing the `Lock` interface or for implementing synchronizer utilities like the `CountDownLatch` class of `java.util.concurrent`. Subclasses must define `tryAcquire( )`, `tryRelease( )`, `tryAcquireShared( )`, `tryReleaseShared( )`, and `isHeldExclusively`.

Figure 16-104. java.util.concurrent.locks.AbstractQueuedSynchronizer


```
public abstract class AbstractQueuedSynchronizer implements Serializable {
// Protected Constructors
     protected AbstractQueuedSynchronizer( );
// Nested Types
     public class ConditionObject implements Condition, Serializable;
// Public Instance Methods
     public final void acquire(int arg);
     public final void acquireInterruptibly(int arg) throws InterruptedException;
     public final void acquireShared(int arg);
     public final void acquireSharedInterruptibly(int arg) throws InterruptedException;
     public final java.util.Collection<Thread> getExclusiveQueuedThreads( );
     public final Thread getFirstQueuedThread( );
     public final java.util.Collection<Thread> getQueuedThreads( );
     public final int getQueueLength( );
     public final java.util.Collection<Thread> getSharedQueuedThreads( );
     public final java.util.Collection<Thread> getWaitingThreads(AbstractQueuedSynchronizer.
        ConditionObject condition);
     public final int getWaitQueueLength(AbstractQueuedSynchronizer.ConditionObject condition);
     public final boolean hasContended( );
     public final boolean hasQueuedThreads( );
     public final boolean hasWaiters(AbstractQueuedSynchronizer.ConditionObject condition);
     public final boolean isQueued(Thread thread);
     public final boolean owns(AbstractQueuedSynchronizer.ConditionObject condition);
     public final boolean release(int arg);
     public final boolean releaseShared(int arg);
     public final boolean tryAcquireNanos(int arg, long nanosTimeout)
        throws InterruptedException;
     public final boolean tryAcquireSharedNanos(int arg, long nanosTimeout)
        throws InterruptedException;
// Public Methods Overriding Object
     public String toString( );
// Protected Instance Methods
     protected final boolean compareAndSetState(int expect, int update);
     protected final int getState( );
     protected boolean isHeldExclusively( );
     protected final void setState(int newState);
     protected boolean tryAcquire(int arg);
     protected int tryAcquireShared(int arg);
     protected boolean tryRelease(int arg);
     protected boolean tryReleaseShared(int arg);
}
```

AbstractQueuedSynchronizer.ConditionObject java.util.concurrent.locks

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 5.0 ***serializable***

This class implements the `Condition` interface and is suitable for use with an `AbstractQueuedSynchronizer`.

```
public class AbstractQueuedSynchronizer.ConditionObject implements Condition, Serializable {
// Public Constructors
     public ConditionObject( );
// Methods Implementing Condition
     public final void await( ) throws InterruptedException;
     public final boolean await(long time, java.util.concurrent.TimeUnit unit)
        throws InterruptedException;
     public final long awaitNanos(long nanosTimeout) throws InterruptedException;
     public final void awaitUninterruptibly( );
     public final boolean awaitUntil(java.util.Date deadline) throws InterruptedException;
     public final void signal( );
     public final void signalAll( );
// Protected Instance Methods
     protected final java.util.Collection<Thread> getWaitingThreads( );
     protected final int getWaitQueueLength( );
     protected final boolean hasWaiters( );
}
```

Passed To

`AbstractQueuedSynchronizer.{getWaitingThreads( ), getWaitQueueLength( ),hasWaiters( ),owns( )}`

Condition **java.util.concurrent.locks**

Java 5.0

This interface defines an alternative to the `wait( )`, `notify( )`, and `notifyAll( )` methods of `java.lang.Object`. `Condition` objects are always associated with a corresponding `Lock`. Obtain a `Condition` with the `newCondition( )` method of `Lock`.

There are five choices for waiting. The no-argument version of `await( )` is the simplest: it blocks until the thread is signaled or interrupted. `awaitUninterruptibly( )` blocks until the thread is signaled and ignores interrupts. The other three waiting methods are timed waits: they all wait until signaled, interrupted, or until the specified time elapses. `await( )` and `awaitUntil( )` return `true` if they are signaled and `false` if a timeout occurs. `awaitNanos( )` specifies the timeout in nanoseconds. It returns zero or a negative number if the timeout elapses. If it wakes up because of a signal (or because of a spurious wakeup), it returns an estimate of the time remaining in the timeout. If it turns out that the thread needs to continue waiting, this return value can be used as the new timeout value.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

The `signal( )` and `signalAll( )` methods are just like the `notify( )` and `notifyAll( )` methods of `Object`. `signal( )` wakes up one waiting thread, and `signalAll( )` wakes up all waiting threads.

Locking considerations apply to the use of a `Condition` object just as they apply to the use of the `wait( )` and `notify( )` methods of `Object`. Before a thread can call any of the waiting or signaling methods of a `Condition`, it must hold the `Lock` associated with the condition. When the thread begins waiting, it automatically relinquishes the `Lock`, and when it awakes because of a signal, timeout, or interrupt, it must reacquire the lock before it can proceed. A thread is guaranteed to hold the lock when it returns from one of the waiting methods.

Threads waiting on a `Condition` may wake up spuriously, just as they may when waiting on an `Object`. Therefore, calls to wait on a `Condition` are typically written in the form of a loop so that the desired condition is retested when the thread wakes up.

```
public interface Condition {
// Public Instance Methods
     void await( ) throws InterruptedException;
     boolean await(long time, java.util.concurrent.TimeUnit unit)
        throws InterruptedException;
     long awaitNanos(long nanosTimeout) throws InterruptedException;
     void awaitUninterruptibly( );
     boolean awaitUntil(java.util.Date deadline) throws InterruptedException;
     void signal( );
     void signalAll( );
}
```

Implementations

`AbstractQueuedSynchronizer.ConditionObject`

Passed To

`ReentrantLock.{getWaitingThreads( ),getWaitQueueLength( ), hasWaiters( )},ReentrantReadWriteLock.{getWaitingThreads( ), getWaitQueueLength( ),hasWaiters( )}`

Returned By

`Lock.newCondition( ),ReentrantLock.newCondition( ), ReentrantReadWriteLock.ReadLock.newCondition( ), ReentrantReadWriteLock.WriteLock.newCondition( )`

Lock java.util.concurrent.locks

Java 5.0

This interface represents a flexible API for preventing thread concurrency with locking. `Lock` defines four methods for acquiring a lock. The simplest method is `lock( )` which

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

blocks indefinitely and uninterruptibly until the lock is acquired. This method is similar to entering a `synchronized` block. `lockInterruptibly( )` blocks until the lock is acquired or until the thread is interrupted. The no-argument version of `tryLock( )` acquires the lock and returns `true` if the lock is currently available or returns `false` without blocking if the lock is unavailable. The two-argument version of `tryLock( )` is a timed method: it blocks until it acquires the lock (in which case it returns `true`), or until the specified timeout elapses (in which case it returns `false`), or until the thread is interrupted (in which case it throws `InterruptedException`).

Once a `Lock` has been acquired, no other thread can acquire it until it is released with the `unlock( )` method. In order to ensure that locks are always released, even in the presence of unanticipated exceptions, it is typical to begin a `try` block immediately after acquiring the lock and to call `unlock( )` from the associated `finally` clause.

Obtain a `Condition` object associated with a `Lock` by calling `newCondition( )`. See `Condition` for details. See `ReentrantLock` for a concrete implementation of the `Lock` interface.

```
public interface Lock {
// Public Instance Methods
     void lock( );
     void lockInterruptibly( ) throws InterruptedException;
     Condition newCondition( );
     boolean tryLock( );
     boolean tryLock(long time, java.util.concurrent.TimeUnit unit)
        throws InterruptedException;
     void unlock( );
}
```

Implementations

`ReentrantLock`, `ReentrantReadWriteLock.ReadLock`, `ReentrantReadWriteLock.WriteLock`

Returned By

`ReadWriteLock.{readLock( ),writeLock( )}`, `ReentrantReadWriteLock.{readLock( ),writeLock( )}`

LockSupport java.util.concurrent.locks

Java 5.0

This class provides a low-level alternative to the deprecated methods `Thread.suspend( )` and `Thread.resume( )`. The `park( )`, `parkNanos( )`, and `parkUntil( )` methods suspend, or park, the thread until it is unparked by another thread with `unpark( )`, or until it is interrupted by another thread, or until the specified

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

time elapses. `parkNanos( )` parks the thread for the specified number of nanoseconds. `parkUntil( )` parks the thread until the specified time, using the millisecond representation of `System.currentTimeMillis( )`. Any call to these parking methods may return spuriously, so it is important to call `park( )` in a loop that can repark the thread if it should not have resumed.

Unpark a thread with the `unpark( )` method. Note that while the parking methods affect the current thread, the `unpark( )` method affects the thread you specify. If the specified thread is not parked, the next time that thread calls one of the `park( )` methods, it returns immediately instead of blocking.

```
public class LockSupport {
// No Constructor
// Public Class Methods
     public static void park( );
     public static void parkNanos(long nanos);
     public static void parkUntil(long deadline);
     public static void unpark(Thread thread);
}
```

ReadWriteLock java.util.concurrent.locks

Java 5.0

This interface represents a pair of `Lock` objects with special locking behavior that is useful for concurrent algorithms in which reader threads frequently access a data structure and writer threads only infrequently modify the structure. The `Lock` returned by `readLock( )` may be locked by multiple threads at the same time as long as no thread has the `writeLock( )` locked. See `ReentrantReadWriteLock` for a concrete implementation with implementation-specific locking details.

```
public interface ReadWriteLock {
// Public Instance Methods
     Lock readLock( );
     Lock writeLock( );
}
```

Implementations

`ReentrantReadWriteLock`

ReentrantLock java.util.concurrent.locks

Java 5.0 ***serializable***

This class implements the `Lock` interface and adds instrumentation methods to determine what thread currently holds the lock, to return the number of threads waiting to acquire the lock or waiting on an associated `Condition`, and to test whether a specified thread is waiting to acquire the lock.

The name of this class includes the term "reentrant" because the thread that holds the lock can call any of the locking methods again, and they return immediately without blocking. `isHeldByCurrentThread( )` tests whether the current thread already holds the lock. `getHoldCount( )` returns the number of times that the current thread has acquired this lock. `unlock( )` must be called this number of times before the lock is actually relinquished.

A "fair" lock may be created by passing `true` to the `ReentrantLock( )` constructor. If you do this, the lock will always be granted to the thread that has been waiting for it the longest.

Figure 16-105. java.util.concurrent.locks.ReentrantLock


```
public class ReentrantLock implements Lock, Serializable {
// Public Constructors
     public ReentrantLock( );
     public ReentrantLock(boolean fair);
// Public Instance Methods
     public int getHoldCount( );                                          default:0
     public final int getQueueLength( );                                  default:0
     public int getWaitQueueLength(Condition condition);
     public final boolean hasQueuedThread(Thread thread);
     public final boolean hasQueuedThreads( );
     public boolean hasWaiters(Condition condition);
     public final boolean isFair( );                                      default:false
     public boolean isHeldByCurrentThread( );                             default:false
     public boolean isLocked( );                                          default:false
// Methods Implementing Lock
     public void lock( );
     public void lockInterruptibly( ) throws InterruptedException;
     public Condition newCondition( );
     public boolean tryLock( );
     public boolean tryLock(long timeout, java.util.concurrent.TimeUnit unit)
        throws InterruptedException;
     public void unlock( );
// Public Methods Overriding Object
     public String toString( );
// Protected Instance Methods
     protected Thread getOwner( );
     protected java.util.Collection<Thread> getQueuedThreads( );
     protected java.util.Collection<Thread> getWaitingThreads(Condition condition);
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

ReentrantReadWriteLock **java.util.concurrent.locks**

Java 5.0 ***serializable***

This class implements the `ReadWriteLock` interface. The locks returned by the `readLock( )` and `writeLock( )` methods are instances of the inner classes `ReadLock` and `WriteLock`. `ReentrantReadWriteLock` defines a "fair mode" and includes instrumentation methods like `ReentrantLock` does.

Any number of threads can acquire the read lock as long as no thread holds or is attempting to acquire the write lock. When a thread attempts to acquire the write lock, no new read locks are granted. When all existing readers have relinquished the lock, the writer acquires the lock, and no reads are allowed until the writer has relinquished it. A thread that holds the write lock may downgrade to a read lock by acquiring the read lock and then relinquishing the write lock.

Because the read lock is not exclusive, it cannot have a `Condition` associated with it. The `ReadLock.newCondition( )` method throws `UnsupportedOperationException`.

Figure 16-106. java.util.concurrent.locks.ReentrantReadWriteLock


```
public class ReentrantReadWriteLock implements ReadWriteLock, Serializable {
// Public Constructors
     public ReentrantReadWriteLock( );
     public ReentrantReadWriteLock(boolean fair);
// Nested Types
     public static class ReadLock implements Lock, Serializable;
     public static class WriteLock implements Lock, Serializable;
// Public Instance Methods
     public final int getQueueLength( );                                    default:0
     public int getReadLockCount( );                                        default:0
     public int getWaitQueueLength(Condition condition);
     public int getWriteHoldCount( );                                       default:0
     public final boolean hasQueuedThread(Thread thread);
     public final boolean hasQueuedThreads( );
     public boolean hasWaiters(Condition condition);
     public final boolean isFair( );                                        default:false
     public boolean isWriteLocked( );                                       default:false
     public boolean isWriteLockedByCurrentThread( );                        default:false
     public ReentrantReadWriteLock.ReadLock readLock( );
     public ReentrantReadWriteLock.WriteLock writeLock( );
// Public Methods Overriding Object
     public String toString( );
// Protected Instance Methods
     protected Thread getOwner( );
     protected java.util.Collection<Thread> getQueuedReaderThreads( );
     protected java.util.Collection<Thread> getQueuedThreads( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    protected java.util.Collection<Thread> getQueuedWriterThreads( );
    protected java.util.Collection<Thread> getWaitingThreads(Condition condition);
}
```

Passed To

`ReentrantReadWriteLock.ReadLock.ReadLock( )`,
`ReentrantReadWriteLock.WriteLock.WriteLock( )`

ReentrantReadWriteLock.ReadLock java.util.concurrent.locks

Java 5.0 ***serializable***

A `Lock` implementation for reader threads. Any number of threads can acquire the lock as long as the corresponding `WriteLock` is not held. `newCondition( )` throws `UnsupportedOperationException`.

```
public static class ReentrantReadWriteLock.ReadLock implements Lock, Serializable {
// Protected Constructors
    protected ReadLock(ReentrantReadWriteLock lock);
// Methods Implementing Lock
    public void lock( );
    public void lockInterruptibly( ) throws InterruptedException;
    public Condition newCondition( );
    public boolean tryLock( );
    public boolean tryLock(long timeout, java.util.concurrent.TimeUnit unit)
       throws InterruptedException;
    public void unlock( );
// Public Methods Overriding Object
    public String toString( );
}
```

Returned By

`ReentrantReadWriteLock.readLock( )`

ReentrantReadWriteLock.WriteLock java.util.concurrent.locks

Java 5.0 ***serializable***

A `Lock` implementation for writer threads. This lock can be acquired only when all holders of the corresponding `ReadLock` have relinquished the locks. While this lock is held, no other thread may acquire either this lock or the corresponding `ReadLock`.

```
public static class ReentrantReadWriteLock.WriteLock implements Lock, Serializable {
// Protected Constructors
    protected WriteLock(ReentrantReadWriteLock lock);
// Methods Implementing Lock
    public void lock( );
    public void lockInterruptibly( ) throws InterruptedException;
    public Condition newCondition( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public boolean tryLock( );
    public boolean tryLock(long timeout, java.util.concurrent.TimeUnit unit)
       throws InterruptedException;
    public void unlock( );
// Public Methods Overriding Object
    public String toString( );
}
```

Returned By

`ReentrantReadWriteLock.writeLock( )`

Package java.util.jar

Java 1.2

The `java.util.jar` package contains classes for reading and writing Java archive, or JAR, files. 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.

Interfaces

```
public interface Pack200.Packer;
public interface Pack200.Unpacker;
```

Collections

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


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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;
public abstract class Pack200;
```

Exceptions

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

Attributes java.util.jar

Java 1.2 ***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`.

Figure 16-107. java.util.jar.Attributes


```
public class Attributes implements java.util.Map<Object,Object>, Cloneable {
// Public Constructors
     public Attributes( );
     public Attributes(java.util.jar.Attributes attr);
     public Attributes(int size);
// Nested Types
     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<java.util.Map.Entry<Object,Object>> entrySet( );
     public boolean equals(Object o);
     public Object get(Object name);
     public int hashCode( );
     public boolean isEmpty( );                          default:true
     public java.util.Set<Object> keySet( );
     public Object put(Object name, Object value);
     public void putAll(java.util.Map<?,?> attr);
     public Object remove(Object name);
     public int size( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public java.util.Collection<Object> values( );
// Public Methods Overriding Object
    public Object clone( );
// Protected Instance Fields
    protected java.util.Map<Object,Object> map;
}
```

Returned By

`java.net.JarURLConnection.{getAttributes( ),getMainAttributes( )}, JarEntry.getAttributes( ),Manifest.{getAttributes( ), getMainAttributes( )}`

Attributes.Name java.util.jar

Java 1.2

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( )`

JarEntry java.util.jar

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.2 ***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 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. In Java 5.0, this digital signature information may be more conveniently retrieved as an array of `CodeSigner` objects.

Figure 16-108. java.util.jar.JarEntry


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

Returned By

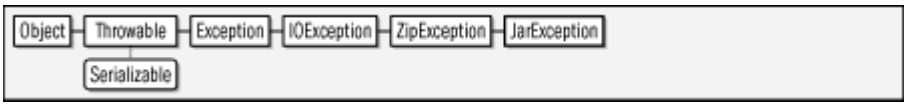
`java.net.JarURLConnection.getJarEntry( )`, `JarFile.getJarEntry( )`, `JarInputStream.getNextJarEntry( )`

JarException **java.util.jar**

Java 1.2 ***serializable checked***

Signals an error while reading or writing a JAR file.

Figure 16-109. java.util.jar.JarException




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

```
    public JarException(String s);
}
```

JarFile java.util.jar

Java 1.2

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. As of 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.

Figure 16-110. java.util.jar.JarFile


```
public class JarFile extends java.util.zip.ZipFile {
// Public Constructors
     public JarFile(String name) throws java.io.IOException;
     public JarFile(java.io.File file) throws java.io.IOException;
     public JarFile(String name, boolean verify) throws java.io.IOException;
     public JarFile(java.io.File file, boolean verify) throws java.io.IOException;
1.3  public JarFile(java.io.File file, boolean verify, int mode) throws java.io.IOException;
// Public Constants
     public static final String MANIFEST_NAME;                  ="META-INF/MANIFEST.MF"
// Public Instance Methods
     public JarEntry getJarEntry(String name);
     public Manifest getManifest( ) throws java.io.IOException;
// Public Methods Overriding ZipFile
     public java.util.Enumeration<JarEntry> entries( );
     public java.util.zip.ZipEntry getEntry(String name);
     public java.io.InputStream getInputStream(java.util.zip.ZipEntry ze)
        throws java.io.IOException;     synchronized
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Passed To

`Pack200.Packer.pack( )`

Returned By

`java.net.JarURLConnection.getJarFile( )`

JarInputStream java.util.jar

Java 1.2 ***closeable***

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`.

Figure 16-111. java.util.jar.JarInputStream


```
public class JarInputStream extends java.util.zip.ZipInputStream {
// Public Constructors
     public JarInputStream(java.io.InputStream in) throws java.io.IOException;
     public JarInputStream(java.io.InputStream in, boolean verify) throws java.io.IOException;
// Public Instance Methods
     public Manifest getManifest( );
     public JarEntry getNextJarEntry( ) throws java.io.IOException;
// Public Methods Overriding ZipInputStream
     public java.util.zip.ZipEntry getNextEntry( ) throws java.io.IOException;
     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);
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Passed To

`Pack200.Packer.pack( )`

JarOutputStream — java.util.jar

Java 1.2 — ***closeable flushable***

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`.

Figure 16-112. java.util.jar.JarOutputStream


```
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;
}
```

Passed To

`Pack200.Unpacker.unpack( )`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Manifest java.util.jar

Java 1.2 ***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.

Figure 16-113. java.util.jar.Manifest


```
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<String,java.util.jar.Attributes> 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( )`

Returned By

`java.net.JarURLConnection.getManifest( )`, `JarFile.getManifest( )`, `JarInputStream.getManifest( )`

Pack200 java.util.jar

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 5.0

This class is a factory for creating `Pack200.Packer` and `Pack200.Unpacker` objects for compressing JAR files to Pack200 archives and for uncompresssing those archives back into JAR files.

```
public abstract class Pack200 {
// No Constructor
// Nested Types
     public interface Packer;
     public interface Unpacker;
// Public Class Methods
     public static Pack200.Packer newPacker( );                    synchronized
     public static Pack200.Unpacker newUnpacker( );
}
```

Pack200.Packer java.util.jar

Java 5.0

This interface defines the API for an object that can convert a JAR file to an output stream in Pack200 (or gzipped Pack200) format. Obtain a `Packer` object with the `Pack200.newPacker( )` factory method. Configure the packer before using it by setting properties in the `Map` returned by the `properties( )` method. The constants defined by this class represent the names (and in some cases values) of properties that can be set. Pack a JAR file by passing `JarFile` or `JarInputStream` to a `pack( )` method along with the byte output stream to which the packed representation should be written. You can monitor the progress of the packer engine by querying the `PROGRESS` property in the `properties( )` map. The value is the completion percentage as an integer between 0 and 100 (or -1 to indicate a stall or error.) If you want to be notified of changes to the `PROGRESS` property, register a `java.beans.PropertyChangeListener` with `addPropertyChangeListener( )`. See also the *pack200* command in Chapter 8.

```
public interface Pack200.Packer {
// Public Constants
     public static final String CLASS_ATTRIBUTE_PFX;              ="pack.class.attribute."
     public static final String CODE_ATTRIBUTE_PFX;               ="pack.code.attribute."
     public static final String DEFLATE_HINT;                    ="pack.deflate.hint"
     public static final String EFFORT;                          ="pack.effort"
     public static final String ERROR;                           ="error"
     public static final String FALSE;                           ="false"
     public static final String FIELD_ATTRIBUTE_PFX;             ="pack.field.attribute."
     public static final String KEEP;                            ="keep"
     public static final String KEEP_FILE_ORDER;                 ="pack.keep.file.order"
     public static final String LATEST;                          ="latest"
     public static final String METHOD_ATTRIBUTE_PFX;            ="pack.method.attribute."
     public static final String MODIFICATION_TIME;               ="pack.modification.time"
     public static final String PASS;                            ="pass"
     public static final String PASS_FILE_PFX;                   ="pack.pass.file."
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public static final String PROGRESS;                                ="pack.progress"
    public static final String SEGMENT_LIMIT;                           ="pack.segment.limit"
    public static final String STRIP;                                   ="strip"
    public static final String TRUE;                                    ="true"
    public static final String UNKNOWN_ATTRIBUTE;                       ="pack.unknown.attribute"
// Event Registration Methods (by event name)
    void addPropertyChangeListener(java.beans.PropertyChangeListener listener);
    void removePropertyChangeListener(java.beans.PropertyChangeListener listener);
// Public Instance Methods
    void pack(JarInputStream in, java.io.OutputStream out) throws java.io.IOException;
    void pack(JarFile in, java.io.OutputStream out) throws java.io.IOException;
    java.util.SortedMap<String,String> properties( );
}
```

Returned By

`Pack200.newPacker( )`

Pack200.Unpacker java.util.jar

Java 5.0

This interface defines an API for converting a file or stream in Pack200 (or gzipped Pack200) format into a JAR file in the form of a `JarOutputStream`. Obtain an `Unpacker` object with the `Pack200.newUnpacker( )` method. Before using an unpacker, you may configure it by setting properties in the `Map` returned by the `properties( )` method. Unpack a JAR file with the `unpack( )` method, specifying a `File` or stream of packed bytes. Monitor the progress of the unpacker by querying the `PROGRESS` key in the `Map` returned by `properties( )`. The value should be an `Integer` representing a completion percentage between 0 and 100. If you want to be notified of changes to the `PROGRESS` property, register a `java.beans.PropertyChangeListener` with `addPropertyChangeListener( )`. See also the *unpack200* command in Chapter 8.

```
public interface Pack200.Unpacker {
// Public Constants
    public static final String DEFLATE_HINT;                            ="unpack.deflate.hint"
    public static final String FALSE;                                   ="false"
    public static final String KEEP;                                    ="keep"
    public static final String PROGRESS;                                ="unpack.progress"
    public static final String TRUE;                                    ="true"
// Event Registration Methods (by event name)
    void addPropertyChangeListener(java.beans.PropertyChangeListener listener);
    void removePropertyChangeListener(java.beans.PropertyChangeListener listener);
// Public Instance Methods
    java.util.SortedMap<String,String> properties( );
    void unpack(java.io.InputStream in, JarOutputStream out) throws java.io.IOException;
    void unpack(java.io.File in, JarOutputStream out) throws java.io.IOException;
}
```

Returned By

`Pack200.newUnpacker( )`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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 pertinant 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. In Java 5.0, `LoggingMXBean` defines an interface for monitoring and management of the logging facility through the `javax.management` packages (which are beyond the scope of this book).

Interfaces

```
public interface Filter;
public interface LoggingMXBean;
```

Classes

```
public class ErrorManager;
public abstract class Formatter;
   public class SimpleFormatter extends Formatter;
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
   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.util.logging

Java 1.4

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 table below 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

Figure 16-114. java.util.logging.ConsoleHandler


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

ErrorManager java.util.logging

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.4

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 `setErrorManager( )` 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.setErrorManager( )`

Returned By

`Handler.getErrorManager( )`

FileHandler **java.util.logging**

Java 1.4

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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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 below. The constructor arguments are:

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:

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 *`limit`* argument is nonzero and the *`count`* argument is greater than one. `FileHandler` 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 being incremented. When the number of log files reaches the number specifed by *`count`*, 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 obtian 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

Figure 16-115. java.util.logging.FileHandler


```
public class FileHandler extends StreamHandler {
// Public Constructors
     public FileHandler( ) throws java.io.IOException, SecurityException;
     public FileHandler(String pattern) throws java.io.IOException, SecurityException;
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    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.util.logging

Java 1.4

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
    boolean isLoggable(LogRecord record);
}
```

Passed To

`Handler.setFilter( ),Logger.setFilter( )`

Returned By

`Handler.getFilter( ),Logger.getFilter( )`

Formatter java.util.logging

Java 1.4

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 one of the pre-defined 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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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);                    synchronized
     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.util.logging

Java 1.4

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. 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 it 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`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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

Subclasses

`MemoryHandler`, `StreamHandler`

Passed To

`java.util.logging.Formatter.{getHead( ),getTail( )}`, `Logger.{addHandler( ),removeHandler( )}`, `MemoryHandler.MemoryHandler( )`, `XMLFormatter.{getHead( ),getTail( )}`

Returned By

`Logger.getHandlers( )`

Level java.util.logging

Java 1.4 ***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`, `FINER`, 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.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-116. java.util.logging.Level


```
public class Level implements Serializable {
// Protected Constructors
     protected Level(String name, int value);
     protected Level(String name, int value, String resourceBundleName);
// 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( )`, `Logger.getLevel( )`, `LogRecord.getLevel( )`, `MemoryHandler.getPushLevel( )`

Logger java.util.logging

Java 1.4

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:

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

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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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 threadsafe 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
// 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 Filter getFilter( );
     public Handler[ ] getHandlers( );                      synchronized
     public Level getLevel( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public String getName( );
    public Logger getParent( );
    public java.util.ResourceBundle getResourceBundle( );
    public String getResourceBundleName( );
    public boolean getUseParentHandlers( );                          synchronized
    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 setFilter(Filter newFilter) throws SecurityException;
    public void setLevel(Level newLevel) throws SecurityException;
    public void setParent(Logger parent);
    public void setUseParentHandlers(boolean useParentHandlers);       synchronized
    public void severe(String msg);
    public void throwing(String sourceClass, String sourceMethod, Throwable thrown);
    public void warning(String msg);
}
```

Passed To

`LogManager.addLogger( )`

Returned By

`LogManager.getLogger( )`

LoggingMXBean java.util.logging

Java 5.0

This interface defines the API for the `javax.management` "management bean" for the logging system. Obtain an instance with the static method `LogManager.getLoggingMXBean( )`. The methods of this class allow the monitoring of all registered loggers and their logging level and allow management to change the logging level of any named logger.

```
public interface LoggingMXBean {
// Public Instance Methods
     String getLoggerLevel(String loggerName);
     java.util.List<String> getLoggerNames( );
     String getParentLoggerName(String loggerName);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    void setLoggerLevel(String loggerName, String levelName);
}
```

Returned By

`LogManager.getLoggingMXBean( )`

LoggingPermission java.util.logging

Java 1.4 ***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 adminstrators configuring security policies may need to be familiar with it.

Figure 16-117. java.util.logging.LoggingPermission


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

LogManager java.util.logging

Java 1.4

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 heirarchical names; and (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.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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 specifing 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 configuation 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), and then register those `Handler` objects on the root `Logger`, where they are inherited by all other loggers. (We'll see more about the root logger below.) 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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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 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.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
public class LogManager {
// Protected Constructors
     protected LogManager( );
// Public Constants
5.0  public static final String LOGGING_MXBEAN_NAME;     ="java.util.logging:type=Logging"
// Public Class Methods
5.0  public static LoggingMXBean getLoggingMXBean( );                synchronized
     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<String> 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;
}
```

LogRecord java.util.logging

Java 1.4 ***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.

Figure 16-118. java.util.logging.LogRecord


```
public class LogRecord implements Serializable {
// Public Constructors
     public LogRecord(Level level, String msg);
// Public Instance Methods
     public Level getLevel( );
     public String getLoggerName( );
     public String getMessage( );
     public long getMillis( );
     public Object[ ] getParameters( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
        public java.util.ResourceBundle getResourceBundle( );
        public String getResourceBundleName( );
        public long getSequenceNumber( );
        public String getSourceClassName( );
        public String getSourceMethodName( );
        public int getThreadID( );
        public Throwable getThrown( );
        public void setLevel(Level level);
        public void setLoggerName(String name);
        public void setMessage(String message);
        public void setMillis(long millis);
        public void setParameters(Object[ ] parameters);
        public void setResourceBundle(java.util.ResourceBundle bundle);
        public void setResourceBundleName(String name);
        public void setSequenceNumber(long seq);
        public void setSourceClassName(String sourceClassName);
        public void setSourceMethodName(String sourceMethodName);
        public void setThreadID(int threadID);
        public void setThrown(Throwable thrown);
    }
```

Passed To

`ConsoleHandler.publish( ),FileHandler.publish( ), Filter.isLoggable( ),java.util.logging.Formatter.{format( ), formatMessage( )},Handler.{isLoggable( ),publish( )}, Logger.log( ),MemoryHandler.{isLoggable( ),publish( )}, SimpleFormatter.format( ),SocketHandler.publish( ),StreamHandler. {isLoggable( ),publish( )},XMLFormatter.format( )`

MemoryHandler java.util.logging

Java 1.4

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 whenver 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`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

properties inherited from `Handler`. The table below 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	1000 log records
`pushLevel`	java.util.logging.MemoryHandler.push	Level.SEVERE

Figure 16-119. java.util.logging.MemoryHandler


```
public class MemoryHandler extends Handler {
// Public Constructors
     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.util.logging

Java 1.4

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

Figure 16-120. java.util.logging.SimpleFormatter


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


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

SocketHandler java.util.logging

Java 1.4

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 table below 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

Figure 16-121. java.util.logging.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.util.logging

Java 1.4

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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`.

Figure 16-122. java.util.logging.StreamHandler


```
public class StreamHandler extends Handler {
// Public Constructors
     public StreamHandler( );
     public StreamHandler(java.io.OutputStream out, java.util.logging.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.util.logging

Java 1.4

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.

Figure 16-123. java.util.logging.XMLFormatter

```
public class XMLFormatter extends java.util.logging.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 persistant 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 code below 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 we 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));
  }
}
```

Interfaces

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


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Events

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

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.util.prefs

Java 1.4

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.

Figure 16-124. java.util.prefs.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);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

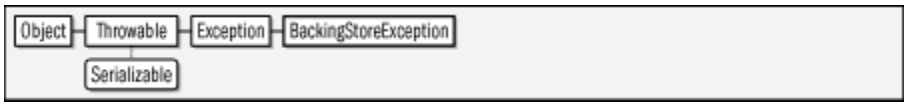
```
    public boolean nodeExists(String path) throws BackingStoreException;
    public Preferences parent( );
    public void put(String key, String value);
    public void putBoolean(String key, boolean value);
    public void putByteArray(String key, byte[ ] value);
    public void putDouble(String 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 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;
}
```

BackingStoreException java.util.prefs

Java 1.4 ***serializable checked***

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.

Figure 16-125. java.util.prefs.BackingStoreException




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


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Thrown By

Too many methods to list.

InvalidPreferencesFormatException java.util.prefs

Java 1.4 ***serializable checked***

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.

Figure 16-126. java.util.prefs.InvalidPreferencesFormatException


```
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.util.prefs

Java 1.4 ***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.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-127. java.util.prefs.NodeChangeEvent


```
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.util.prefs

Java 1.4 ***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.

Figure 16-128. java.util.prefs.NodeChangeListener


```
public interface NodeChangeListener extends java.util.EventListener {
// Public Instance Methods
     void childAdded(NodeChangeEvent evt);
     void childRemoved(NodeChangeEvent evt);
}
```

Passed To

`AbstractPreferences.{addNodeChangeListener( ), removeNodeChangeListener( )},Preferences.{addNodeChangeListener( ),removeNodeChangeListener( )}`

PreferenceChangeEvent java.util.prefs

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.4 ***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.

Figure 16-129. java.util.prefs.PreferenceChangeEvent


```
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.util.prefs**

Java 1.4 ***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.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-130. java.util.prefs.PreferenceChangeListener


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

Passed To

`AbstractPreferences.{addPreferenceChangeListener( ), removePreferenceChangeListener( )}, Preferences.{addPreferenceChangeListener( ), removePreferenceChangeListener( )}`

Preferences java.util.prefs

Java 1.4

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 persistant "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 below), and a 8192-character limit on preference value strings.

`Preferences` does not have a public construtor. To obtain a `Preferences` object for use in your application, you must must use one of the static methods described below.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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. `absoutePath( )` 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.

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 recieve 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`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`put( )` and the various type-specific `put...( )` convenience methods may return asynchonously, before the new preference value is stored persistantly 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 persistantly. (Note that it is not necessary to call `flush( )` before an application terminates: all preferences will eventually be made persistant.) 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;
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    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( ), PreferencesFactory.{systemRoot( ),userRoot( )}`

PreferencesFactory java.util.prefs

Java 1.4

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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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
     Preferences systemRoot( );
     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 `java.lang.CharSequence` of text, and defines various methods for matching the pattern to the text. In Java 5.0, the `MatchResult` interface represents the result of a match. `Matcher` implements this interface and can be queried directly.

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.

Interfaces

```
public interface MatchResult;
```

Classes

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

Exceptions

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

Matcher java.util.regex

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.4

A `Matcher` objects encapsulate 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. In Java 5.0, you can use a new `Pattern` object on the current character sequence with the `usePattern( )` method.

Once you have created or reset a `Matcher`, there are three types of comparisons you can perform between the regular expression and the character sequence. All three comparisons operate on the current *region* of the character sequence. By default, this region is the entire sequence. In Java 5.0, however, you can set the bound of the region with `region( )`. The simplest type of comparison is the `matches( )` method. It returns `true` if the pattern matches the complete region of the character sequence, and returns `false` otherwise. The `lookingAt( )` method is similar: it returns true if the pattern matches the complete region, or if it matches some subsequence at the beginning of the region. If the pattern does not match the start of the region, `lookingAt( )` returns `false`. `matches( )` requires the pattern to match both the beginning and ending of the region, 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 region. 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 `MatchResult` interface defines the `start( )`, `end( )` and `group( )` methods that return the starting position, the ending position and the text of the match, and of any matching subexpressions within the `Pattern`. See `MatchResult` for details. The `MatchResult` interface is new in Java 5.0, but `Matcher` implements all of its methods in Java 1.4 as well. Calling `MatchResult` methods on a `Matcher` returns results from the most recent match. If you want to store these results, call `toMatchResult( )` to obtain an indepedent, immutable `MatchResult` object whose methods can be queried later.

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 region. The first time `find( )` is called after a `Matcher` is created or after the `reset( )` method is called, it starts it 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 `MatchResult` 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, preceed it with a backslash. In Java 5.0, you can use the static `quoteReplacement( )` method to properly quote any special characters in a replacement string so that the string will be interpreted literally.

`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 `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`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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 begining 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( )`.

Prior to Java 5.0, the region of the input text that a `Matcher` operates on is the entire character sequence. In Java 5.0, you can define a different region with the `region( )` method, which specifies the position of the first character in the region and the position of the first character after the end of the region. `regionStart( )` and `regionEnd( )` return the current value of these region bounds. By default, regions are "anchoring" which means that the start and end of the region match the `^` and `$` anchors. (See `Pattern` for regular expression grammar details.) Call `useAnchoringBounds( )` to turn anchoring bounds on or off in Java 5.0. The bounds of a region are "opaque" by default, which means that the `Matcher` will not look through the bounds in an attempt to match look-ahead or look-behind assertions (see `Pattern`). In Java 5.0, you can make the bounds transparent with `useTransparentBounds(true)`.

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

Figure 16-131. java.util.regex.Matcher


```
public final class Matcher implements MatchResult {
// No Constructor
// Public Class Methods
5.0  public static String quoteReplacement(String s);
// Public Instance Methods
     public Matcher appendReplacement(StringBuffer sb, String replacement);
     public StringBuffer appendTail(StringBuffer sb);
     public int end( );                                  Implements:MatchResult
     public int end(int group);                        Implements:MatchResult
     public boolean find( );
     public boolean find(int start);
     public String group( );                             Implements:MatchResult
     public String group(int group);                   Implements:MatchResult
     public int groupCount( );                           Implements:MatchResult
5.0  public boolean hasAnchoringBounds( );
5.0  public boolean hasTransparentBounds( );
5.0  public boolean hitEnd( );
     public boolean lookingAt( );
     public boolean matches( );
     public Pattern pattern( );
5.0  public Matcher region(int start, int end);
5.0  public int regionEnd( );
5.0  public int regionStart( );
     public String replaceAll(String replacement);
     public String replaceFirst(String replacement);
5.0  public boolean requireEnd( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public Matcher reset( );
    public Matcher reset(CharSequence input);
    public int start( );                              Implements:MatchResult
    public int start(int group);                     Implements:MatchResult
5.0 public MatchResult toMatchResult( );
5.0 public Matcher useAnchoringBounds(boolean b);
5.0 public Matcher usePattern(Pattern newPattern);
5.0 public Matcher useTransparentBounds(boolean b);
// Methods Implementing MatchResult
    public int end( );
    public int end(int group);
    public String group( );
    public String group(int group);
    public int groupCount( );
    public int start( );
    public int start(int group);
// Public Methods Overriding Object
5.0 public String toString( );
}
```

Returned By

`Pattern.matcher( )`

MatchResult java.util.regex

Java 5.0

This interface represents the results of a regular expression matching operation performed by a `Matcher`. `Matcher` implements this interface directly, and you can use the methods defined here to obtain the results of the most recent match performed by a `Matcher`. You can also save those most recent match results in a separate immutable `MatchResult` object by calling the `toMatchResult( )` method of the `Matcher`.

The no-argument versions of the `start( )` and `end( )` method return the index of the first character that matched the pattern and the index of the last character that matched plus one (the index of the first character following the matched text), respectively. Some regular expressions can match the empty string. If this occurs, `end( )` returns the same value as `start( )`. The no-argument version of `group( )` returns the text that matched the pattern.

If the matched `Pattern` includes capturing subexpressions within parentheses, the other methods of this interface provide details about the text that matched each of those subexpressions. Pass a group number to `start( )`, `end( )`, or `group( )` to obtain the start, end, or text that matched the specified group. `groupCount( )` returns the number of subexpressions. Groups are numbered from 1, however, so legal group numbers run from 1 to the value returned by `groupCount( )`. Groups 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. Group 0 represents the

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

entire regular expression, so passing 0 to `start( )`, `end( )`, or `group( )` is the same as calling the no-argument version of the method.

```
public interface MatchResult {
// Public Instance Methods
     int end( );
     int end(int group);
     String group( );
     String group(int group);
     int groupCount( );
     int start( );
     int start(int group);
}
```

Implementations

`Matcher`

Returned By

`java.util.Scanner.match( )`, `Matcher.toMatchResult( )`

Pattern java.util.regex

Java 1.4 ***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,

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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:

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. The table below, is a quick-reference for regular expression 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". In Java 5.0, the static `quote( )` method quotes all special characters in a string so that you can match arbitrary text literally without worrying that punctuation in that text will be interpreted specially. For complete details on regular expressions see a book like *Programming Perl* by Larry Wall et. al., or *Mastering Regular Expressions* by Jeffrey E. F. Friedl.

Table 16-3. Java regular expression quick reference

Syntax	Matches
Single characters	
x	The character *x*, as long as *x* is not a punctuation character with special meaning in the regular expression syntax.
p	The punctuation character *p*.
\\	The backslash character
\n	Newline character \u000A.
\t	Tab character \u0009.
\r	Carriage return character \u000D.
\f	Form feed character \u000C.
\e	Escape character \u001B.
\a	Bell (alert) character \u0007.
\u*xxxx*	Unicode character with hexadecimal code *xxxx*.
\x*xx*	Character with hexadecimal code *xx*.
\0*n*	Character with octal code *n*.
\0*nn*	Character with octal code *nn*.
\0*nnn*	Character with octal code *nnn*, where *nnn* <= 377.
\c*x*	The control character ^*x*.
Character classes	
[...]	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 below.
[^...]	Any one character not between the brackets.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Syntax	Matches
`[a-z0-9]`	Character range: a character between (inclusive) `a` and `z` or `0` and `9`.
`[0-9[a-fA-F]]`	Union of classes: same as `[0-9a-fA-F]`
`[a-z&&[aeiou]]`	Intersection of classes: same as `[aeiou]`.
`[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, then it matches any character including line terminators.
`\d`	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`	ASCII word character: `[a-zA-Z0-9_]`.
`\W`	Anything but ASCII word characters: `[^\w]`.
`\p{`*group*`}`	Any character in the named group. See group names below. 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}`	ASCII lowercase letter: `[a-z]`.
`\p{Upper}`	ASCII uppercase: `[A-Z]`.
`\p{ASCII}`	Any ASCII character: `[\x00-\x7f]`.
`\p{Alpha}`	ASCII letter: `[a-zA-Z]`.
`\p{Digit}`	ASCII digit: `[0-9]`.
`\p{XDigit}`	Hexadecimal digit: `[0-9a-fA-F]`.
`\p{Alnum}`	ASCII letter or digit: `[\p{Alpha}\p{Digit}]`.
`\p{Punct}`	ASCII punctuation: one of `!"#$%& ( )*+,-./:;<=>?@[\]^_ {|}~]`.
`\p{Graph}`	visible ASCII character: `[\p{Alnum}\p{Punct}]`.
`\p{Print}`	visible ASCII character: same as `\p{Graph}`.
`\p{Blank}`	ASCII space or tab: `[ \t]`.
`\p{Space}`	ASCII whitespace: `[ \t\n\f\r\x0b]`.
`\p{Cntrl}`	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 `java.lang.Character` for a set of constants that correspond to these subcategories; however, note that the full set of one- and two-letter codes is not documented in this book.
`\p{`*block*`}`	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: `\p{InOgham}` or `\p{InMathematicalOperators}`. See `java.lang.Character.UnicodeBlock` for a list of Unicode block names.
Sequences, alternatives, groups, and references	
xy	Match *x* followed by *y*.
x`|`*y*	Match *x* or *y*.
`(...)`	Grouping. Group subexpression within parentheses into a single unit that can be used with `*`, `+`, `?`, `|`, and so on. Also "capture" the characters that match this group for use later.
`(?:...)`	Grouping only. Group subexpression as with `( )`, but do not capture the text that matched.
`\`*n*	Match the same characters that were matched when capturing group number *n* was first matched. Be careful when *n* is followed by another digit: the largest number that is a valid group number will be used.
Repetition[1]	

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Syntax	Matches
x?	zero or one occurrence of *x*; i.e., *x* is optional.
*x**	zero or more occurrences of *x*.
x+	one or more occurrences of *x*.
x{*n*}	exactly *n* occurrences of *x*.
x{*n*,}	*n* or more occurrences of *x*.
x{*n*,*m*}	at least *n*, and at most *m* occurrences of *x*.
Anchors[2]	
^	The beginning of the input string, or if the MULTILINE flag is specified, the beginning of the string or of any new line.
$	The end of the input string, or if the MULTILINE flag is specified, the end of the string or of line within the string.
\b	A word boundary: a position in the string between a word and a nonword character.
\B	A position in the string that is not a word boundary.
\A	The beginning of the input string. Like ^, but never matches the beginning of a new line, regardless of what flags are set.
\Z	The end of the input string, ignoring any trailing line terminator.
\z	The end of the input string, including any line terminator.
\G	The end of the previous match.
(?=*x*)	A positive look-ahead assertion. Require that the following characters match *x*, but do not include those characters in the match.
(?!*x*)	A negative look-ahead assertion. Require that the following characters do not match the pattern *x*.
(?<=*x*)	A positive look-behind assertion. Require that the characters immediately before the position match *x*, but do not include those characters in the match. *x* must be a pattern with a fixed number of characters.
(?<!*x*)	A negative look-behind assertion. Require that the characters immediately before the position do not match *x*. *x* must be a pattern with a fixed number of characters.
Miscellaneous	
(?>*x*)	Match *x* 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.
(?*onflags*-*offflags*)	Don t match anything, but turn on the flags specified by *onflags*, and turn off the flags specified by *offflags*. These two strings are combinations in any order of the following letters and correspond to the following Pattern constants: i (CASE_INSENSITIVE), d (UNIX_LINES), m (MULTILINE), s (DOTALL), u (UNICODE_CASE), and x (COMMENTS). 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.
(?*onflags*-*offflags*:*x*)	Match *x*, applying the specified flags to this subexpression only. This is a noncapturing group, like (?:...), with the addition of flags.
\Q	Don't match anything, but quote all subsequent pattern text until \E. All characters within such a quoted section are interpreted as literal characters to match, and none (except \E) have special meanings.
\E	Don't match anything; terminate a quote started with \Q.
#*comment*	If the COMMENT flag is set, pattern text between a # and the end of the line is considered a comment and is ignored.

[1] 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 quantifiers above with a question mark. For example, use *? instead of *, and use {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."

[2] 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.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-132. java.util.regex.Pattern


```
public final class Pattern implements Serializable {
// No Constructor
// Public Constants
     public static final int CANON_EQ;                                    =128
     public static final int CASE_INSENSITIVE;                            =2
     public static final int COMMENTS;                                    =4
     public static final int DOTALL;                                      =32
5.0  public static final int LITERAL;                                  =16
     public static final int MULTILINE;                                   =8
     public static final int UNICODE_CASE;                                =64
     public static final int UNIX_LINES;                                  =1
// Public Class Methods
     public static Pattern compile(String regex);
     public static Pattern compile(String regex, int flags);
     public static boolean matches(String regex, CharSequence input);
5.0  public static String quote(String s);
// Public Instance Methods
     public int flags( );
     public Matcher matcher(CharSequence input);
     public String pattern( );
     public String[ ] split(CharSequence input);
     public String[ ] split(CharSequence input, int limit);
// Public Methods Overriding Object
5.0  public String toString( );
}
```

Passed To

`java.util.Scanner.{findInLine( ),findWithinHorizon( ),hasNext( ),next( ),skip( ),useDelimiter( )},Matcher.usePattern( )`

Returned By

`java.util.Scanner.delimiter( ),Matcher.pattern( )`

PatternSyntaxException java.util.regex

Java 1.4 ***serializable unchecked***

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.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-133. java.util.regex.PatternSyntaxException


```
public class PatternSyntaxException extends IllegalArgumentException {
// Public Constructors
     public PatternSyntaxException(String desc, String regex, int index);
// Public Instance Methods
     public String getDescription( );
     public int getIndex( );
     public String getPattern( );
// Public Methods Overriding Throwable
     public String getMessage( );
}
```

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 implements ZipConstants;
public class Inflater;
public class InflaterInputStream extends java.io.FilterInputStream;
   public class GZIPInputStream extends InflaterInputStream;
   public class ZipInputStream extends InflaterInputStream implements ZipConstants;
public class ZipEntry implements Cloneable, ZipConstants;
public class ZipFile implements ZipConstants;
```

Exceptions

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


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Adler32 java.util.zip

Java 1.1

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.

Figure 16-134. java.util.zip.Adler32


```
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.util.zip

Java 1.1 ***closeable***

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.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-135. java.util.zip.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.util.zip

Java 1.1 ***closeable flushable***

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.

Figure 16-136. java.util.zip.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.util.zip

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.1

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
     long getValue( );
     void reset( );
     void update(int b);
     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.util.zip

Java 1.1

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.

Figure 16-137. java.util.zip.CRC32

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
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.util.zip

Java 1.1 ***serializable checked***

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

Figure 16-138. java.util.zip.DataFormatException


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

Thrown By

`Inflater.inflate( )`

Deflater java.util.zip

Java 1.1

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
// 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 int getAdler( );                                  synchronized default:1
5.0  public long getBytesRead( );                          synchronized default:0
5.0  public long getBytesWritten( );                       synchronized default:0
     public int getTotalIn( );                               default:0
     public int getTotalOut( );                              default:0
     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.util.zip

Java 1.1 — ***closeable flushable***

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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.

Figure 16-139. java.util.zip.DeflaterOutputStream


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

Subclasses

`GZIPOutputStream`, `ZipOutputStream`

GZIPInputStream — java.util.zip

Java 1.1 — ***closeable***

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.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-140. java.util.zip.GZIPInputStream


```
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.util.zip

Java 1.1 ***closeable flushable***

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.

Figure 16-141. java.util.zip.GZIPOutputStream


```
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.util.zip

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.1

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);
// Public Instance Methods
     public void end( );                                    synchronized
     public boolean finished( );                            synchronized
     public int getAdler( );                                synchronized default:1
5.0  public long getBytesRead( );                         synchronized default:0
5.0  public long getBytesWritten( );                      synchronized default:0
     public int getRemaining( );                            synchronized default:0
     public int getTotalIn( );                            default:0
     public int getTotalOut( );                           default:0
     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.util.zip

Java 1.1 ***closeable***

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`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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.

Figure 16-142. java.util.zip.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;
5.0  public void mark(int readlimit);                          synchronized empty
5.0  public boolean markSupported( );                                  constant
     public int read( ) throws java.io.IOException;
     public int read(byte[ ] b, int off, int len) throws java.io.IOException;
5.0  public void reset( ) throws java.io.IOException;                  synchronized
     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.util.zip

Java 1.1 ***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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

own `ZipEntry` objects and initialize them to contain the entry name and other appropriate information before writing the contents of the entry.

Figure 16-143. java.util.zip.ZipEntry


```
public class ZipEntry implements Cloneable, ZipConstants {
// 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
// Public Instance Methods
     public String getComment( );
     public long getCompressedSize( );
     public long getCrc( );
     public byte[ ] getExtra( );
     public int getMethod( );
     public String getName( );
     public long getSize( );
     public long getTime( );
     public boolean isDirectory( );
     public void setComment(String comment);
1.2  public void setCompressedSize(long csize);
     public void setCrc(long crc);
     public void setExtra(byte[ ] extra);
     public void setMethod(int method);
     public void setSize(long size);
     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( ), ZipFile.getInputStream( ), ZipOutputStream.putNextEntry( )`

Returned By

`java.util.jar.JarFile.getEntry( ), java.util.jar.JarInputStream.{createZipEntry( ), getNextEntry( )}, ZipFile.getEntry( ), ZipInputStream.{createZipEntry( ), getNextEntry( )}`

ZipException java.util.zip

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.1 ***serializable checked***

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

Figure 16-144. java.util.zip.ZipException


```
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.util.zip**

Java 1.1

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

Figure 16-145. java.util.zip.ZipFile

```
public class ZipFile implements ZipConstants {
// 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<? extends ZipEntry> 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.util.zip

Java 1.1 — ***closeable***

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.

Figure 16-146. java.util.zip.ZipInputStream


```
public class ZipInputStream extends InflaterInputStream implements ZipConstants {
// Public Constructors
     public ZipInputStream(java.io.InputStream in);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
// 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.util.zip

Java 1.1 — ***closeable flushable***

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( )`, where 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.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 16-147. java.util.zip.ZipOutputStream


```
public class ZipOutputStream extends DeflaterOutputStream implements ZipConstants {
// 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.JarOutput Stream`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01