Contenedores de objetos: Collection, Map y Array

Índice

1.Introducción al Collection FrameWork

Un marco de trabajo es un conjunto de interfaces y/o clases proporcionados para resolver un problema determinado. La intención es utilizar las clases proporcionadas, extenderlas o implementar los interfaces.

En java.util contamos con diversos interfaces e implementaciones de los mismos que nos permiten agrupar a otros objetos mediante una estructura de datos cuyo tamaño es modificable. Las arrays no permiten variación en el número de sus elementos pero a diferencia de los primeros , los cuales solo pueden contener elementos del tipo Object, su componente puede ser de cualquier tipo.

Los interfaces denominados “interfaces de Colección básicos” (core collection interfaces) son Collection, Map y sus derivados: List, Set y SortedSet son subinterfaces del primero, mientras que SortedMap deriva del segundo.

Las implementaciones de propósito general son clases que implementan estos interfaces. ArrayList y LinkedList implementan List, HashSet implementa Set, TreeSet implementa SortedSet. Map es implementado por HashMap y SortedMap por TreeMap. Normalmente el tipo de las variables que se declaran es uno de los interfaces y no el de estas implementaciones. Por ejemplo: List l = new ArrayList().

De esta forma si posteriormente desea cambiarse la implementación a LinkedList el código escrito para el tipo del interfaz funcionará sin cambio alguno.

Vector, Stack y Hashtable son clases existentes en Java 1.1. En Java 1.2 se denominan legacy implementations (implementaciones heredadas). Todas ellas están sincronizadas -a diferencia de la s clases nuevas-. Vector sufrió una transformación en la cual le fueron añadidos métodos para implementar List. Hashtable, en cambio, implementa Map.

Existen unas clases en java.util que empiezan por Abstract + nombre de interface. Estas son implementaciones parciales de dichos interfaces. Las implementaciones de propósito general derivan de ellas. Su intención es proporcionar un punto de partida a la hora de construir nuestras propias implementaciones de los interfaces básicos, evitando la implementación de todos los métodos definidos en ellos.

Todos los métodos que añaden o eliminan elementos en los interfaces básicos son opcionales. Es decir, las clases que implementan dichos interfaces podrían optar por no implementarlos. El principal objetivo del marco de trabajo Colección fue la simplicidad para facilitar su aprendizaje. La existencia de métodos opcionales evita una proliferación de interfaces en el marco de trabajo. Por ejemplo, el método java.util.Arrays.asList(Object[]) devuelve un objeto del tipo List. Esta lista es implementada por la array pasada como argumento. Por lo tanto no pueden añadirse o eliminarse elementos de esta lista, puesto que no es posible variar el tamaño de una matriz. La lista retornada no es realmente del tipo List, puesto que la invocación de los métodos opcionales sobre dicha lista produce una excepción runtime del tipo UnsupportedOperationException. Sin la utilización de métodos opcionales esta lista debería ser de algún otro tipo como por ejemplo un inexistente FixedSizeList.

Todas las implementaciones de propósito general implementan los métodos opcionales. También los implementan Vector y Hashtable.

Una colección es inmodificable si no puede sufrir variación en el número de sus elementos. Y es inmutable si además, ninguno de sus elementos puede variar tras su creación, en el sentido de ser reemplazado en la colección por otro objeto.

Un objeto del tipo java.util.Iterator proporciona los elementos contenidos en una colección en una secuencia, dependiendo del contenedor no siempre predecible. Todos los Iterator obtenidos de las implementaciones generales y las heredadas poseen la característica de fallo-rapido: Si la colección o mapa del cual se obtuvo el iterator es modificada estructuralmente (varia el número de elementos) por un método no definido en el propio iterator, el próximo mensaje enviado al iterator provocará una excepción cuyo tipo es ConcurrentModificationException. De esta manera, se impide la utilización de un iterator que había sido invalidado por la acción de un método no declarado en él sobre la estructura de datos que hasta entonces representaba. Todas las implementaciones, incluyendo las heredadas cuentan con un método toString capaz de imprimir el contenido de la colección o mapa.

Algunas implementaciones pueden forzar restricciones sobre sus elementos. Por ejemplo, no admitir null.

El intento de incorporación de un elemento que viole una restricción provocará una excepción. Pero, también lo hará la eliminación (obviamente el elemento no esta en el contenedor) y el test de presencia.

2.Métodos hashCode y equals en clase Object

Debido a la importancia de estos métodos en los interfaces e implementaciones del marco de trabajo Colección, serán tratados aquí.

public int hashCode()

A la hora de acceder, añadir, o eliminar un objeto contenido en un hashtable, la implementación de dicha estructura invocará este método sobre el objeto para obtener un int que pueda ser utilizado en la elaboración del indice en el hashtable.

Durante la ejecución de un programa este método ha de retornar el mismo int siempre que sea invocado sobre el mismo objeto. Siempre y cuando sea factible ha de ser único para cada objeto. Por ello, aunque esta implementación no es requerida, el método devuelve la dirección física del objeto en memoria.

Si es obligatorio que si a.equals(b) retorna verdadero se cumpla a.hashCode() == b.hashCode(), pero no lo es que si devuelve falso sean distintos; aunque la velocidad del hashtable sería mayor si se cumple.

Esto conlleva sobrescribir el método hashCode cuando se sobrescriba equals.

No lanza ninguna excepción.

public boolean equals(Object)

Retorna verdadero si el objeto que recibe el mensaje (sobre el cual se invoca el método) es equivalente al especificado como argumento.

Esta relación es :

reflexiva a.equals(a)

simétrica a.equals(b) == b.equals(a)

transitiva a.equals(b) == b.equals(c) == true implica a.equals(c) == true

La relación debe mantenerse siempre que el estado de los objetos sobre los que se basa la comparación en la implementación del método no varíe.

En la clase Object este método retorna verdadero si dos variables apuntan al mismo objeto. Esta comparación es idéntica a la realizada por “==”.

No lanza ninguna excepción. Devuelve falso si el argumento es nulo.

3.Interfaces Comparable y Comparator

java.lang.Comparable

Se dice que las clases que implementan este interfaz cuentan con un “orden natural”. Este orden es total. Esto significa que siempre han de poder ordenarse dos objetos cualesquiera de la clase que implementa este interfaz.

Si una clase implementa Comparable puede utilizarse en las siguientes situaciones sin necesidad de un Comparator:

* Si es una matriz puede ser ordenada mediante Arrays.sort(unaArray).
* Si es del tipo List puede ser ordenada mediante Collections.sort(List), o puede buscarse en ella mediante Collections.binarySearch(List, Object) (Object también ha de implementarlo).
* Puede ser utilizada como un elemento dentro de un TreeSet, o clave (key) en un TreeMap.

Es muy recomendable aunque no requerido que el “orden natural sea consistente con equals”. Esto se cumple si dos objetos que fueron considerados igualmente ordenados por la implementación de este interfaz son determinados equivalentes por equals. Según la API los SortedSet o SortedMap cuyos elementos o claves no posean un orden natural consistente con equals podrían comportarse de forma irregular, además incumplen el contrato que Set o Map impusieron en base al método equals.

Casi todos las clases básicas de Java (core classes) que implementan Comparable lo hacen en una forma consistente con equals. Una excepción es java.math.Bigdecimal cuyo orden natural establece que dos objetos BigDecimal con el mismo valor pero diferente precisión (4.0 y 4.00) son ordenados igualmente. Comparable define un único método int compareTo(Object) que devuelve un número negativo, cero o positivo si el objeto que recibe el mensaje es menor, igual, o mayor que el objeto pasado como argumento. La API especifica que se implemente de la forma descrita. Esto impone un orden ascendente. Para lograr un orden descendente tan solo ha de devolver positivo cuando devolvía negativo y viceversa.

Este método debe arrojar una ClassCastException si el tipo del argumento impide la comparación con el tipo del objeto sobre el que se invoco el método. Normalmente suele comprobarse que sus clases sean idénticas. He aquí un ejemplo de implementación (Name.java) y del orden natural de la clase String (TestName.java):

La clase String implementan este método de forma que las mayúsculas anteceden a las minúsculas. Si deseamos una ordenación alfabética deberemos recurrir a un Comparator que utilice el método String.compareToIgnoreCase().

Las clases que implementan este interfaz son String, Byte, Short, Long, Integer, Character, Float, Double, Date, File, CollationKey, BigInteger, BigDecimal y ObjectStreamField.

```
//Name.java
import java.util.*;

public class Name implements Comparable {
  private String  firstName, lastName;

  public Name(String firstName, String lastName) {
    if (firstName==null || lastName==null)
      throw new NullPointerException();//Comprobar los nulos
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public String firstName(){return firstName;}
  public String lastName() {return lastName;}

  public boolean equals(Object o) {
    if (!(o instanceof Name))
      return false;//método equals retorna false si argumento
    Name n = (Name)o;//no es del tipo adecuado o es nulo
    return n.firstName.equals(firstName) &&
n.lastName.equals(lastName);
  }

  public int hashCode() {
    return 31*firstName.hashCode() + lastName.hashCode();
  }//hashcode tiene en cuenta ambos campos

  public String toString() {return firstName + " " + lastName;}

  public int compareTo(Object o) {
    Name n = (Name)o;//arroja ClassCastException si argumento no
    int lastCmp = lastName.compareTo(n.lastName);//es del tipo Name
    return (lastCmp!=0 ? lastCmp :
```

```
    firstName.compareTo(n.firstName));
    //primero comparamos los apellidos y si son iguales
    //los nombres
  }
}

//TestName.java
import java.util.*;

public class TestName {
  public static void main(String[] args) {
    TreeSet ts = new TreeSet();
    ts.add(new Name("José", "Botella"));
    ts.add(new Name("josé", "Botella"));
    ts.add(new Name("jose", "Botella"));
    ts.add(new Name("Jose", "Botella"));
    ts.add(new Name("Llongeras", "Nuñez"));
    ts.add(new Name("Jesús", "López"));
    ts.add(new Name("Conchi", "Díaz"));
    ts.add(new Name("Emilio", "Botella"));
    ts.add(new Name("Josefa", "Nuñez"));
    ts.add(new Name("manuela", "Llanos"));
    ts.add(new Name("Manuela", "Pérez"));
    System.out.println(ts);
  }
}
/*
C:\Java\TIJ2\ch9>java  TestName
[Emilio Botella, Jose Botella, JosÚ Botella,
jose Botella, josÚ Botella, Conchi DÝaz,
manuela Llanos, Jes·s L¾pez, Josefa Nu±ez,
Llongeras Nu±ez, Manuela PÚrez]

Primero se ordenan por apellidos y después por nombres.

El orden natural de la clase String ordena los carácteres
según su código Unicode:
May-min-acentos
Nota:La salida en DOS produce unos caráteres diferentes
para las vocales acentuadas
*/
```

java.util.Comparator

Si una clase ya tiene ordenación natural y deseamos una ordenación descendente, o bien distinta en cuanto los campos del objeto considerados para la ordenación, o simplemente requerimos diversas formas de ordenar una clase, haremos que una clase distinta de aquella que va ser ordenada implemente este interfaz. Comparator impone un orden total. Objetos cuyos tipos implementen este interfaz pueden ser utilizados en las siguientes situaciones para especificar un orden distinto al natural:

* Como argumento a un constructor TreeSet o TreeMap
* Collections.sort(List, Comparator) , Arrays.sort(Object[], Comparator) y
* Collections.binarySearch(List, Object, Comparator) (Object también ha de implementarlo).

Es muy recomendable que el orden impuesto por este interfaz sea consistente con equals: si la implementación de este interfaz determina que dos objetos tienen el mismo orden equals debe considerarlos equivalentes. También es conveniente que la clase que implemente Comparator implemente a su vez java.io.Serializable; de otra forma el TreeMap o TreeSet no sería serializable.

Este interfaz define dos métodos:

a)boolean equals(Object)
Indica si dos objetos Comparator establecen el mismo orden. Solo es necesario su implementación por motivos de rendimiento en determinados programas. Normalmente no se implementa pues todo objeto hereda la implementación de la clase Object.b)int compare(Object, Object)
Devuelve un int negativo, cero o positivo si el primer objeto es menor, igual o mayor que el segundo.Si se implementa de esta manera asegura un orden ascendente. Si devuelve positivo, cero o negativo respectivamente impone un orden descendente. La API especifica que se implemente para un orden ascendente.El implementador debe asegurarse que:
b1)sgn(compare(x, y) == -sgn(compare(y, x)) para todas las w, y. Esto implica que compare(x, y) lanzará una excepción solo si compare(y, x) la lanza.

b2)la relación es transitiva: “compare(x, y) > 0” && “compare(y, z) > 0” implican “compare(x, z) > 0”

b3)”compare(x, y) == 0” implica “sgn(compare(x, z) == sgn(compare(y, z))” para todas las z.Este método lanzará ClassCastException si el tipo de los argumentos impide la comparación por este Comparator.
La clase java.util.Collections tiene un método estático, reverseOrder(), que produce un Comparator que impone un orden natural descendente. Su implementación simplemente cambia el signo que una implementación normal retornaría.

Sigue un ejemplo de una ordenación alfabética en Ingles (TestAlfabeticComparator.java) y una ordenación alfabética en Español (ComparadorEspanol.java)

```
//TestAlfabeticComparator.java
import java.util.*;

public class TestAlfabeticComparator {
  public static void main(String[] args) {
    TreeSet ts = new TreeSet(new AlfabeticComparator());
    ts.add("José");
    ts.add("josé");
    ts.add("jose");
    ts.add("Jose");
    ts.add("Llongeras");
    ts.add("Jesús");
    ts.add("Conchi");
    ts.add("Emilio");
    ts.add("Nuñez");
    ts.add("manuela");
    ts.add("Manuela");
    System.out.println(ts);
  }
}

class AlfabeticComparator implements Comparator {
  public int compare(Object o1, Object o2) {
   return ((String) o1).compareToIgnoreCase((String) o2);
   //Compara cadenas ignorando las may/min
  }
}

/*
C:\Java\TIJ2\ch9>java   TestAlfabeticComparator
[Conchi, Emilio, Jes·s, jose, JosÚ, Llongeras,
manuela, Nu±ez]

José y josé son la misma sin consideración de las
may/mins luego la 2ª no se añade José y jose
difieren luego la 2ª se añade jose y Jose son la
misma luego la 2ª no se añade
```

```
*/

//ComparadorEspanol.java
import java.util.*;
import java.text.Collator;

public class ComparadorEspanol {
  public static void main(String[] args) {
    TreeSet ts = new TreeSet(new Comparador(Collator.IDENTICAL));
    //tiene en cuenta todas las diferencias
    ts.add("José");
    ts.add("josé");
    ts.add("jose");
    ts.add("Jose");
    ts.add("Llongeras");
    ts.add("Jesús");
    ts.add("Conchi");
    ts.add("Emilio");
    ts.add("Nuñez");
    ts.add("manuela");
    ts.add("Manuela");
    System.out.println("Orden segun codigo Unicode" + "\n" + ts);
    TreeSet ts2 = new TreeSet(new Comparador(Collator.PRIMARY));
    //no tiene en cuenta ninguna
    ts2.addAll(ts);
    System.out.println("Orden sin considerar may/min o acentos" +
    "\n" +ts2);
  }
}

class Comparador implements Comparator {
  Collator col = Collator.getInstance();
  //establece la fuerza del colador:
  //las diferencias significativas
  Comparador(int Strength) {col.setStrength(Strength);}
  public int compare(Object o1, Object o2) {
    return col.compare((String) o1, (String) o2);
  }
}

/*
C:\Java\TIJ2\ch9>java  ComparadorEspañol
Orden segun codigo Unicode
[Conchi, Emilio, Jes·s, jose, Jose, josÚ, JosÚ,
Llongeras, manuela, Manuela, Nu±ez]
Orden sin considerar may/min o acentos
[Conchi, Emilio, Jes·s, jose, Llongeras, manuela,
Nu±ez]
*/
```

4.Utilidades para los objetos contenedores

Las clases “java.util.Arrays” y “java.util.Collections” están compuestas exclusivamente de métodos estáticos que proporcionan utilidades para el manejo de arrays y objetos del tipo Collection.

También veremos el método System.arraycopy()

Nota sobre el sentido de la ordenación impuesto por sort y el sentido en que esta ordenada la lista o matriz que binarySearch examina: La API especifica que en estos dos métodos, tanto en la clase Arrays como en la clase Collections, el sentido de la ordenación sea ascendente. Esto es debido a que las clases que

implementan Comparable o Comparator lo hacen en la forma descrita en la API. Siguiendo estas recomendaciones las “comparadores” que creásemos serían siempre ascendentes. Para lograr una ordenación inversa utilizaríamos el método reverseOrder de la clase Collection. Alternativamente puede implementarse un Comparator o Comparable descendente y utilizarse en ambos métodos. Luego, lo realmente importante es mantener el sentido con el que se ha ordenado un contenedor a la hora de buscar en él.

Es conveniente recordar que los tipos primitivos implementan Comparable en sentido ascendente.

java.util.Arrays

public static List asList(Object[])
Retorna una “vista “ (no una copia) de la matriz pasada como argumento, que puede ser manipulada como si fuera una lista. Como dicha lista es respaldada por una matriz no pueden agregarse o eliminarse elementos a ella. Cualquier modificación estructural (del número de elementos) provoca UnsupportedOperationException debido a la no implementación de los métodos opcionales del interfaz List en la lista retornada. El resto de cambios en la lista son reflejados en la matriz y viceversa. public static boolean equals(unPrimitivo[], elMismoPrimitivo[])
equals(Object[], Object[])
Retornan verdadero si las dos matrices son equivalentes. Esto se cumple si contienen los mismos elementos en el mismo orden. También devuelven verdadero si los dos argumentos son nulos.La comparación entre objetos se realiza mediante o1.equals(o2).Para float: new Float(f1).equals(new Float(f2))(equals se diferencia de == en que devuelve verdadero si f1 y f2 son NaN, y falso si uno es +0.0 y el otro -0.0).Para double: new Double(d1).equals(new Double(d2))(Double.equals se diferencia de == de forma análoga a como lo hace Float.equals).Para el resto de los tipos primitivos la comparación se realiza con ==.public static int binarySearch(tipoPrimitivo[], unPrimitivo)
binarySearch(Object[], Object)
binarySearch(Object[], Object, Comparator)
Busca el segundo argumento en la matriz especificada. La matriz previamente debió haber sido ordenada, bien mediante orden natural, bien con el Comparator suministrado como tercer argumento. El comportamiento no es especificado si la matriz no esta ordenada. Tampoco se especifica cual será el elemento encontrado en caso de existencia de duplicados.

Si el signo del valor retornado es positivo dicho valor es la posición que el elemento buscado ocupa en la matriz. Si fuera negativo indicaría la posición donde insertar el elemento: la posición del primer elemento en la lista mayor que él, o , list.size() si fuera mayor que toda la lista. Aplicando -(insert point) -1 donde insert point es el valor devuelto por el método se obtiene la posición donde insertar el elemento.

```
//TestBinarySearch.java
import java.util.*;

public class TestBinarySearch {
  public static void main(String[] args) {
    Integer[] ia = {new Integer(3), new Integer(4), new Integer(2),
                    new Integer(7), new Integer(1), new Integer(5),
                    new Integer(6)};
    Arrays.sort(ia);
    System.out.println("Arrays.sort(ia):" + Arrays.asList(ia));
    System.out.println("Buscando 2: " + Arrays.binarySearch(ia,
                       new Integer(2)));
    int insertPoint = Arrays.binarySearch(ia, new Integer(9));
    System.out.println("Buscando 9: " + insertPoint);
    System.out.println("Punto de insercion (-insert point) -1 :   -(" +
                       insertPoint + ") -1 = " + ((-insertPoint) -1) );
  }
}

/*
C:\Java\TIJ2\ch9>java  TestBinarySearch
Arrays.sort(ia):[1, 2, 3, 4, 5, 6, 7]
```

Buscando 2: 1
Buscando 9: -8
Punto de insercion (-insert point) -1 : -(-8) -1 = 7
*/

public static void sort(tipoPrimitivo[])
sort(tipoPrimitivo[], Comparator)
sort(Object[])
sort(Object[], Comparator)
En el caso de las matrices de objetos estos métodos ordenan la matriz especificada mediante el orden natural de su elemento, si no se especificó el argumento Comparator, y utilizándolo si se proporcionó.

El algoritmo utilizado en la ordenación de datos primitivos es una modificación del QuickSort que garantiza n*log(n). El utilizado en una matriz de objetos ofrece también las mismas prestaciones pero es un mergesort modificado. Además este último es estable: no se moverán elementos iguales.

Estos métodos están sobrecargados con la adicción de dos int, fromIndex y toIndex: la ordenación se realiza únicamente en un rango, incluyendo los elementos situado en la matriz desde fromIndex hasta, pero, sin incluir, al correspondiente a toIndex. Lanzan ArrayIndexOutOfBoundsException si fromIndex < 0 o toIndex > array.length

IllegalArgumentException si fromIndex > toIndex.
public static void fill(tipoPrimitivo[], unPrimitivo)
void fill(tipoprimitivo[], int fromIndex, int toIndex, unPrimitivo)
void fill(Object[], Object)
void fill(Object[], int fromIndex, int toIndex, Object)
Sustituyen todos los elementos de la matriz por el especificado, o bien, un rango especificado por los int de la forma descrita en el método sort.

System.arraycopy()

Este método estático de la clase System proporciona una forma mucho mas rápida de copiar una matriz que el empleo de bucles. static void arraycopy(Object src, int posS, Object dst, int posD, int num)
Copia los elementos de la matriz src residentes desde posS hasta “posS + num -1” en la matriz dst en la posición posD hasta “posD +num -1” respectivamente.

Si ambas matrices fueran la misma, la copia se comporta como si fuera realizada mediante copia previa a una matriz temporal de tamaño num.

Este método lanza las excepciones:
a)NullPointerexception si dst o src son nulas.
b)ArrayStoreException si se cumple cualquiera de los puntos siguientes:
b1)src o dst apuntan a un objeto distinto de una matriz.
b2)El tipo de una matriz no es compatible con el de la otra.
b3)En el caso de que el tipo de las dos matrices sea de referencia, se lanza dicha excepción cuando el tipo de un elemento contenido en src no pueda ser convertido mediante conversión de asignación al tipo de dst.

Una conversión de asignación permite las siguientes conversiones: identidad (entre tipos idénticos), de ampliación -ambas primitiva y de referencia-, y el tipo especial de conversión de disminución aplicable entre determinados tipos primitivos.

Cuando se lanza ArrayStoreException por este motivo (b3) es posible que se hayan efectuado algunas modificaciones en la matriz dst. Concretamente los elementos anteriores al cual provocó la excepción.
c)IndexOutOfBoundsException cuando cualquiera de los tres argumentos numéricos sea negativo. También si las sumas “posD + num” o “posS + num” exceden el tamaño de sus respectivas matrices.

java.util.Collections

Todos los métodos aquí expuestos retornan o toman como argumentos una List, excepto max, min y enumeration que toman como argumento una Collection.

Existen tres campos estáticos finales de tipo List, Map y Set que fueron inicializados para contener un objeto vacio del tipo correspondiente. Sus nombres son EMPTY_LIST, EMPTY_MAP y EMPTY_SET. Todos son serializables e inmutables.

public static void sort(List)
sort(List, Comparator)
Ordenan la lista, bien de forma ascendente según el orden natural, bien de la forma especificada por Comparator.

La ordenación es estable: los elementos iguales no son desplazados. El algoritmo (mergesort modicado) garantiza un tiempo n*log(n). La lista se copia a una matriz donde es ordenada, después se devuelven las referencias a la lista.

La lista ha de ser modificable aunque no es necesario que también pueda modificarse su tamaño.

Excepciones:
a) ClassCastException. Si un elemento en la lista tiene un tipo que impide su comparación con el resto los interfaces Comparable o Comparator deben lanza esta excepción.

b)UnsupportedOperationException. Si la lista no es modificable. (Su iterator soporta la operación set).
public static int binarySearch(List, Object)
binarySearch(List, Object, Comparator)
Buscan el objeto en la lista. Dicha lista debe estar ordenada de forma ascendente. Esto puede lograrse con un orden natural para el primer método, como por ejemplo sort(List); o bien usando un Comparator para el segundo método: sort(List, Comparator). Los resultados no son especificados si la lista esta desordenada. No se asegura cual será el elemento devuelto si existen duplicados.

Devuelven la posición del elemento en la lista (resultado 0 o positivo). Si no esta en la lista devuelve (-(insert point) -1); donde insert point es la posición del primer elemento en la lista mayor que el buscado, o el tamaño de la lista si el elemento buscado es el mayor de todos.

El método responde en un tiempo n*log(n) para una lista de acceso aleatorio, pero si el tipo es LinkedList el tiempo de búsqueda será lineal.

Excepciones:
a)ClassCastException. Si algún elemento en la lista tiene un tipo que impide su comparación con el resto los interfaces Comparable o Comparator lanzarán esta excepción. El argumento Object también ha de implementar uno de estos interfaces y por lo tanto también puede lanzarlo.

```
//TestingBinarySearch.java
import java.util.*;

public class TestingBinarySearch {
  public static void main(String[] args) {
    List al = new ArrayList();
    for(int i=0; i<10 ; i++) al.add(new Integer(i));
    Collections.reverse(al);
    System.out.println(al);
    System.out.println("buscando 9 en lista al posicion : " +
    Collections.binarySearch(al, new Integer(9)));
    System.out.println("la busqueda no tiene exito pues al no" +
                       "estaba ordenada en forma ascendente y sin" +
                       "embargo la busqueda fue realizada " +
                                    "ascendentemente\n\n");
    ////////////////////////
    al.clear();
    for(int i=0; i< 10; i++) al.add(new MyInt(i));
    System.out.println(al);
    System.out.println("buscando 9 en la lista al posicion : " +
```

```
                                    Collections.binarySearch(al, new MyInt(9),
                    new Compa(Compa.UP)));
    System.out.println("la busqueda tiene exito pues al estaba ordenada" +
                                    "ascendentemente por su creación. Aunque no fue" +
                    "ordenada con un comparador ascendente, no obtenemos" +
                    "error mientras utilicemos un comparador ascendente\n\n");
    ////////////////////////
    Collections.sort(al, new Compa(Compa.DOWN));
    System.out.println(al);
    System.out.println("buscando 9 en la lista al posicion :" +
                                    Collections.binarySearch(al, new MyInt(9),
                    new Compa(Compa.DOWN)));
    System.out.println("la busqueda tiene exito pues la lista se ordeno " +
                                    "descendentemente por un comparador que fue suministrado" +
                    "a binarySearch");
  }
}

class Compa implements Comparator {
  //dependiendo del argumento ordena de forma ascendente o descendente
  static final int UP= 1, DOWN= -1;
  private int sgn;
  Compa(int sgn) { this.sgn=sgn; }
  public int compare(Object o1, Object o2) {
    int i1 = ((MyInt) o1).i;
    int i2 = ((MyInt) o2).i;
    return sgn * ( i1 < i2 ? -1 : ( i1==i2 ? 0 : 1 ) );
  }
}

class MyInt {
  int i;
  MyInt(int i) { this.i=i; }
  public String toString() { return i+""; }
}
```

public static void reverse(List)
Ordena la lista en el sentido inverso de la posición actual de sus elementos (independiente del valor de los elementos). No realiza una ordenación descendente. Se ejecuta en un tiempo lineal.

```
import java.util.*;
//TestReverse.java
public class TestReverse {
  public static void main(String[] args) {
    List al = new ArrayList();
    al.add(new Integer(9));
    al.add(new Integer(8));
    al.add(new Integer(7));
    al.add(new Integer(6));
    al.add(new Integer(5));
    al.add(new Integer(4));
    al.add(new Integer(3));
    al.add(new Integer(2));
    al.add(new Integer(1));
    al.add(new Integer(0));
    System.out.println(al + "\n" + "reversing...");
    Collections.reverse(al);
    System.out.println(al + "\n" + "shuffling...");
    Collections.shuffle(al);
    System.out.println(al + "\n" + "reversing...");
```

```
    Collections.reverse(al);
    System.out.println(al);
  }
}

/*
C:\Java\TIJ2\ch9>java TestReverse
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
reversing...
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
shuffling...
[2, 7, 5, 1, 8, 9, 3, 0, 6, 4]
reversing... funciona incluso si la lista no esta ordenada
[4, 6, 0, 3, 9, 8, 1, 5, 7, 2]
*/
```

public static Comparator reverseOrder()
Devuelve un Comparator que introduce una ordenación inversa a la impuesta por el orden natural de un contenedor de objetos que implementan el interfaz Comparable. El Comparator retornado es serializable.

El Comparator devuelto implementa el método compare de forma que devuelve negado el resultado de compareTo pasándole sus dos argumentos.

Puede utilizarse donde se espere un Comparator: Arrays.sort(list, Collections.reverseOrder())
public static void shuffle(List)
shuffle(List, Random)
Aleatorizan la posición de los elementos en la lista. En el segundo caso utiliza como fuente aleatoria el objeto Random suministrado. Se ejecutan en un tiempo lineal para una lista de acceso aleatorio y cuadratico si la lista es de acceso secuencial.

Lanzan UnsupportedOperationException si el iterator de la lista no soporta la operación set.
public static void fill(List, Object)
Remplaza todos los elementos de la lista por el objeto especificado. No modifica el tamaño de la lista.

Se ejecuta en un tiempo lineal.

Lanza UnsupportedOperationException si el iterator de la lista no soporta la operación set.
public static void copy(dstList, srcList)
Copia los elementos residentes en el segundo argumento en el primero. Si el tamaño de dstList es mayor que el de srcList los restantes elementos no se ven afectados.

Este método se ejecuta en un tiempo lineal.

Lanza IndexOutOfBoundsException si el tamaño de dstList es menor que el de srcList, y UnsupportedOperationException si el iterator de la lista no soporta la operación set.
public static Enumeration enumeration(Collection)
Devuelve un objeto Enumeration de la Collection pasada como argumento. Este método permite que las nuevas colecciones sean tratadas por APIs antiguas escritas para manejar Enumeration solamente.
max(Collection, Comparator)
min(Colelction)
min(Collection, Comparator)
Devuelve el objeto que en la Collection especificada sea el menor, o el mayor de todos, de acuerdo al orden natural de tales objetos, o según el Comparator especificado.public static List nCopies(Int, Object)
Devuelve una List inmutable conteniendo Object tantas veces como especifique Int. Es útil para hacer crecer una lista existente mediante el método List.addAll(Collections.nCopies(Int, Object)) lanza IllegalArgumentException si Int < 0public static Set singleton(Object)
List singletonList(Object)
Map singletonMap(keyObject, valueObject)
Devuelve un contenedor inmutable y serializable conteniendo una sola copia de sus argumentos.public static Collection synchronizedCollection(Collection)

List synchronizedList(List)
Map synchronizedMap(Map)
Set synchronizedSet(Set)
SortedSet synchronizedSortedSet(SortedSet)
SortedMap synchronizedSortedMap(SortedMap)

A partir de Java 1.2 las colecciones de objetos no están sincronizadas por defecto. En cambio Vector y Hastable que proceden de java 1.1 si lo están.

Estos métodos devuelven su argumento pero ya sincronizado. Para garantizar un acceso seguro, respecto los hilos de ejecución, a dichas objetos sincronizados, es imperativo que todo acceso posterior se realice sobre los objetos retornados y no los originales pasados como argumento.

Ademas el comportamiento no esta especificado si se utiliza un Iterator, obtenido de los objetos sincronizados, sin haber sido sincronizado todo acceso al Iterator sobre el propio objeto.

Aún cuando el Iterator se halla obtenido desde una vista de la colección, todo acceso a ese Iterator ha de haber sido sincronizado sobre la colección, no sobre la vista. Ver la Java API para los ejemplos.

El objeto retornado será serializable si el argumento lo era.

Nota sobre equals y hashCode en el primer método: La API especifica que la Set o List devuelta por primer método no pasa los métodos hashCode y equals a los objetos originales (argumento del método), sino que emplean los que fueron definidos en la clase Object. De esta forma se preserva dichos métodos en los objetos. Esto no es así en el resto de los métodos. La consecuencia practica es que mientras dos Set o List sincronizados por el primer método, teniendo el mismo contenido no son iguales según equals y hashCode; si lo eran cuando no estaban sincronizados. En cambio con todos los demás métodos, los objetos no difieren a este respecto antes y después de haber sido sincronizados.

```
import java.util.*;
//TestSynchonizedList.java

public class TestSynchronizedList {
  public static void main(String[] args) {
    List cole1 = new ArrayList();
    List cole2 = new ArrayList();
    for(int i=0; i<10; i++) {
      cole1.add(new Integer(i));
      cole2.add(new Integer(i));
    }
    System.out.println(cole1.equals(cole2));
    System.out.println(cole1.hashCode() + " " + cole2.hashCode());
    ////////////////////////////////////////////////
    cole1 = Collections.synchronizedList(cole1);
    cole2 = Collections.synchronizedList(cole2);
    System.out.println(cole1.equals(cole2));
    System.out.println(cole1.hashCode() + " " + cole2.hashCode());
  }
}

/*
C:\Java\TIJ2\ch9>java TestSynchronizedCollection
Dos Set o List con el mismo contenido son iguales y
tienen idéntico hashCode
true
-1631921466 -1631921466
pero esto no se cumple si sicronizamos las colecciones
false
7474923 3242435
*/
```

public static Collection unmodiafiableCollection(Collection)
List unmodiafiableList(List)
Map unmodifiableMap(Map)
Set unmodifiableSet(Set)
SortedSet unmodifiableSortedSet(SortedSet)
SortedMap unmodifiableSortedMap(SortedMap)
Retornan un objeto de solo lectura respecto al argumento. Cualquier operación que pretenda modificar la referencia devuelta, bien directamente, a través de un iterator o vista, lanzará una UsupportedOperationException. El objeto retornado es serializable si el argumento lo era.

Se aplica la misma nota sobre equals y hashCode que en los métodos synchronized.

5.Interfaz Collection

Este es el interfaz padre de Set y List. El framework no proporciona ninguna implementación suya, sino de sus subtipos. El framework tampoco proporciona una implementación de un "Bag", el cual es un contenedor sin secuencia y con posibilidad de duplicados. Cualquier implementación del mismo debería extender este interfaz directamente.

Este interfaz, como tipo, es utilizado para manipular colecciones cuando se desea una generalidad máxima.

Un interfaz no puede contener constructores y por lo tanto Collection no puede imponer la siguiente premisa, que sin embargo, es cumplida por todas sus implementaciones en el framework:

Ha de proveerse un constructor sin argumentos y otro con un argumento del tipo Collection. Este último construye la colección con los elementos contenidos en dicho argumento.

Métodos que operan sobre un elemento

public boolean add(Object) (opcional)
Asegura que el objeto especificado esta contenido en la colección. Esta ambigüedad permite unas definiciones mas precisas en sus subinterfaces. Devuelve verdadero si la colección se modificó como resultado de la llamada. Devuelve falso si el objeto ya estaba contenido en la colección.

La colección podría imponer ciertas restricciones a los elementos a añadir. Si la colección rehusa la adición de un elemento por cualquier razón distinta de su previa existencia en ella, el método debe lanzar una excepción en lugar de retornar falso. De esta forma se preservan la premisa que establece que si este método retorna el elemento se encuentra en la colección.

Excepciones:

ClassCastException. Si el tipo del argumento no es el adecuado para la adición (infringe una restricción).

IllegalArgumentException. Si la restricción es cualquier otra no relacionada con el tipo del argumento.

UnsupportedOperationException. Si la implementación de este interfaz no incorporó este método.
public boolean remove(Object) (opcional)
Remueve un elemento de la colección tal que ambos argumento y el elemento sean determinados equivalentes por el método equals. Si el argumento es nulo, se removería un elemento nulo.

Retorna verdadero si la colección sufrió modificación como resultado de esta llamada.

Lanza UnsupportedOperationException si la implementación de este interfaz no incluyó este método.
public boolean contains(Object)
Retorna verdadero si el argumento y algún elemento en la colección son determinados equivalentes por equals. Si el argumento es nulo retorna verdadero si existe algún elemento nulo.

Métodos que operan sobre varios elementos

public void clear() (opcional)
Vacía la colección.

Lanza UnsupportedOperationException si la implementación de este interfaz no incluyó este método.
public boolean containsAll(Collection)
Retorna verdadero si todos los elementos de la colección pasada como argumento se encuentran también en esta.public boolean retainsAll(Collection) (opcional)
Elimina de la colección que recibe este mensaje (método) aquellos elementos que no se encuentren en la colección especificada como argumento.

Retorna verdadero si la colección fue modificada como resultado de esta llamada.

Lanza UnsupportedOperationException si la implementación de este interfaz no incluyó este método.
public boolean removeAll(Collection) (opcional)
Elimina de la colección que recibe el mensaje todos los elementos contenidos en la colección especificada.

Retorna verdadero si la colección fue modificada como resultado de esta llamada.

Lanza UnsupportedOperationException si la implementación de este interfaz no incluyó este método.
public boolean addAll(Collection) (opcional)
Añade a la colección que recibe este mensaje todos los elementos contenidos en la colección especificada.

Retorna verdadero si la colección fue modificada como resultado de esta llamada.

El comportamiento de esta operación no esta definido si el argumento es modificado durante el transcurso de la misma.

Lanza UnsupportedOperationException si la implementación de este interfaz no incluyó este método.

Métodos que retornan información sobre el número de elementos

public int size()
Devuelve el número de elementos contenidos en la colección. Si la colección contiene mas elementos que Integer.MAX_VALUE retorna Integer.MAX_VALUE.public boolean isEmpty()
Retorna verdadero si la colección no contiene elementos.

Métodos que devuelven una vista de la colección

En este caso empleamos el termino de vista para designar un objeto que proporciona un medio de acceder a todos los elementos en la colección.

public Iterator iterator()
Retorna un objeto que implementa el interfaz Iterator. No hay garantías sobre el orden en que el iterator retorna sus elementos, a no ser que la implementación en particular las otorgue.public Object[] toArray()
Copia los elementos de esta colección a un matriz de objetos, recién creada, que devuelve. Las modificaciones realizadas en los elementos de la matriz o de la colección se reflejan en la otra, puesto que ambas contienen referencias apuntando a los mismos objetos. Sin embargo si se modifica la colección después de haberle sido enviado este mensaje, la matriz retornada queda invalidada como representante de la colección.

Si el Iterator de la colección retorna los elementos en un orden determinado, este mismo orden fue utilizado para poblar la matriz.
public Object[] toArray(Object[])
Si la matriz especificada como argumento tiene un tamaño suficiente para almacenar los elementos de la la colección, se copian dichos elementos en ella empezando por el indice cero, y la matriz argumento es

a su vez devuelta por el método. Si el tamaño fuera mayor, las posiciones de indice mayores que el tamaño de la colección serían establecidas a nulo.

Si el tamaño del argumento fuera insuficiente se crearía una matriz nueva cuyo tipo es el del argumento.

Por lo demás se aplican las mismas consideraciones que en el método anterior.

Lanza ArrayStoreException si el tipo (durante la ejecución) del argumento no es igual o un supertipo de cada uno de los tipos (durante la ejecución) de los elementos contenidos en la colección.

```
import java.util.*;
//TestToArray.java

public class TestToArray {
  public static void main(String[] args){
    Collection col = new ArrayList();
    StringBuffer s = null;
    for( char c='a'; c <'f'; c++) {
      StringBuffer s2 =  new StringBuffer(new Character(c).toString());
            if (c == 'c') s = s2;//mas tarde lo eliminaremos
            col.add(s2);
    }
    StringBuffer[] sb = (StringBuffer[]) col.toArray( new StringBuffer[7] );
    System.out.println("Antes de la modificacion \n" + col);
    System.out.println(Arrays.asList(sb));
    sb[1].append("yuhuu");
    Iterator it = col.iterator();
    ( (StringBuffer) it.next()).append("yuhuu");
    System.out.println("Depues de las modificaciones de los elementos \n" + col);
    System.out.println(Arrays.asList(sb));
    col.remove(s);//equals the StringBuffer no esta basadado en el contenido
    sb[4] = null;
    System.out.println("Depues de las modificaciones de los contenedores \n" + col);
    System.out.println(Arrays.asList(sb));
  }
}

/*
C:\Java\TIJ2\ch9>java  TestToArray
Antes de la modificacion
[a, b, c, d, e]
[a, b, c, d, e, null, null]
Depues de las modificaciones de los elementos
[ayuhuu, byuhuu, c, d, e]
[ayuhuu, byuhuu, c, d, e, null, null]
Depues de las modificaciones de los contenedores
[ayuhuu, byuhuu, d, e]
[ayuhuu, byuhuu, c, d, null, null, null]

*/
```

hashCode y equals

public int hashCode()
Devuelve el código hash para el objeto colección. Idéntico al declarado en la clase Object.public boolean equals(Object)
Devuelve verdadero cuando dos colecciones son consideradas equivalentes. Idéntico al declarado en la clase Object. Leer la API si se desea implementar una colección que no sea un Set o List.

6.Interfaz Iterator

Iterator mejora a Enumeration de Java 1.1, proporcionando las siguientes ventajas:

Los nombres de los métodos han sido mejorados. Ahora es posible eliminar elementos de la colección mientras se esta iterando. Los objetos Enumeration no son fallo-rápido.

```
import java.util.*;
//TestIterator.java

public class TestIterator {
  public static void main(String[] args){
    Collection col = new ArrayList();
    for( char c='a'; c <'f'; c++)
            col.add(new StringBuffer( new Character(c).toString() ));
    System.out.println("Antes de la eliminacion \n" + col);
    Iterator it = col.iterator();
    it.next();
    it.remove();
    System.out.println("Despues de remove()\n" + col);
    try { it.remove(); }
    catch(IllegalStateException e)
            { System.out.println(e + " producida al invocar remove" +
       "dos veces seguidas sin su correspondiente next");
            }
    System.out.println("Uso normal del iterator");

    while(it.hasNext()) {
      StringBuffer sb = (StringBuffer) it.next();
            System.out.println( sb.append("yuhuu") );
    }
  }
}

/*
C:\Java\TIJ2\ch9>java  TestIterator
Antes de la eliminacion
[a, b, c, d, e]
Despues de remove()
[b, c, d, e]
java.lang.IllegalStateException producida al
invocar remove dos veces seguidas sin su correspondiente next
Uso normal del iterator
byuhuu
cyuhuu
dyuhuu
eyuhuu

*/
```

Iterator tiene un comportamiento no especificado cuando la colección es modificada de alguna forma distinta a su método remove durante una iteración. Aunque la característica de fallo rápido no es impuesta por este interfaz, todas las implementaciones de Collection en la API producen un objeto Iterator que la incorporan: next lanza la excepción ConcurrentModificationexception pero hasNext no.

```
import java.util.*;
//TestFailFast.java

public class TestFailFast {
  public static void main(String[] args) {
    Collection col = new ArrayList();
```

```
    for( char c='a'; c <'f'; c++)
            col.add(new StringBuffer( new Character(c).toString() ));
    System.out.println(col);
    Iterator it = col.iterator();
    boolean firstTime = true;
    while(it.hasNext()) {
      try {System.out.println(it.next());}
            catch(ConcurrentModificationException e)
        {e.printStackTrace();}
            if(firstTime) {
              firstTime = false;
              col.add(new StringBuffer( new Character('f').toString()));
            }
            System.out.println(col.size());
    }

  }
}

/*
Esto muestra que efectivamente next() lanza ConcurrentModificationException
si se modifica la colección por un medio distinto de Iterator.remove() mientras
se esta iterando. Pero también muestra que hasNext() no lo hace, sino que devuelve
siempre true porque next() no completa.

java.util.ConcurrentModificationException
 at java.util.AbstractList$Itr.checkForComodification(Unknown Source)
 at java.util.AbstractList$Itr.next(Unknown Source)
 at TestFailFast.main(TestFailFast.java:12)
6
java.util.ConcurrentModificationException
 at java.util.AbstractList$Itr.checkForComodification(Unknown Source)
 at java.util.AbstractList$Itr.next(Unknown Source)
 at TestFailFast.main(TestFailFast.java:12)
6
java.util.ConcurrentModificationException
 at java.util.AbstractList$Itr.checkForComodification(Unknown Source)
 at java.util.AbstractList$Itr.next(Unknown Source)
 at TestFailFast.main(TestFailFast.java:12)
6
java.util.ConcurrentModificationException
 at java.util.AbstractList$Itr.checkForComodification(Unknown Source)
 at java.util.AbstractList$Itr.next(Unknown Source)
 at TestFailFast.main(TestFailFast.java:12)
6
java.util.ConcurrentModificationException
 at java.util.AbstractList$Itr.checkForComodification(Unknown Source)
 at java.util.AbstractList$Itr.next(Unknown Source)
 at TestFailFast.main(TestFailFast.java:12)
6
etc.............
*/
```

public boolean hasNext()
Devuelve verdadero mientras existan elementos no tratados mediante el método next. public Object next()
Devuelve una referencia al siguiente elemento en la colección.

Lanza NoSuchElementException si se invoca un numero de veces superior al número de elementos existentes en la colección.
public void remove() (opcional)

Elimina de la colección el último elemento retornado por next. Solo puede ser llamado una vez por cada llamada a next, y siempre después de aquel.

Excepciones:

IllegalEstateException. Si no se cumplen las condiciones expuestas para la llamada de remove.

UnsupportedOperationException. Si la implementación de este interfaz no incluyó este método.

7.Interfaz Set

Estipula la no existencia de duplicados. El elemento nulo tampoco puede ser contenido mas de una vez.

Extiende de Collection y no añade ningún método a los allí definidos. Sin embargo, si modifica el contrato impuesto por los métodos add, equals y hashCode. Además, las clases que implementen este interfaz deben imponer a sus constructores la imposibilidad de añadir elementos duplicados.

El método equals de los elementos será invocado para averiguar si el objeto a añadir ya existe.

Debe extremarse la precaución de no modificar el estado de un elemento del Set mientras permanece en él, al menos no los campos utilizados por equals; ya que esto provocaría inconsistencia entre sucesivas llamadas al método. En esta situación el comportamiento del Set no esta definido.

Un Set no puede contenerse a si mismo. Debido a la forma en que hashCode es implementado ello provocaría un bucle infinito.

public boolean add(Object) (opcional)
Devuelve verdadero como consecuencia de la inexistencia del argumento en el Set y su posterior adicción.

Devuelve falso si el argumento ya existiera en la colección, y no lo sobrescribe.

Al método equals del argumento se le pasa un elemento del Set para determinar si son equivalentes y por lo tanto considerarlos duplicados.

Excepciones:

ClassCastException. Será lanzada si el tipo del argumento impide su comparación con un elemento.

IllegalArgumentException. Debe lanzarse si cualquier otra razón distinta del tipo y duplicidad impide la adición del argumento. (Infringe alguna restricción que el Set impone a sus elementos).

UnsupportedOperationException. Si la implementación de este interfaz no incluyó este método.
public boolean equals(Object)
Devuelve verdadero si dos Set son equivalentes. El argumento ha ser un Set con el mismo numero de elementos, y los elementos de un Set han de estar contenidos en el otro.public int hashCode()
Devuelve el valor hash de un Set. Es computado sumando el hash de cada uno de sus elementos. De esta forma se cumple la condición impuesta por el método hashCode en Objtec: si dos Set son considerados equivalentes por equals han de tener el mismo hash.
Los métodos de manipulación de varios elementos en un objeto Set desarrollan operaciones sobre conjuntos:

containsAll comprueba si un conjunto esta contenido en otro.

addAll realiza la unión de dos conjuntos.

retainAll proporciona la intersección.

removeAll devuelve la diferencia asimétrica.

Para lograr la diferencia simétrica basta con restar la intersección de dos conjuntos a su unión.

8.Interfaz List

Establece estipulaciones adicionales a las del interfaz Collection en los métodos add, equals, hashCode, iterator y remove. También añade nuevos métodos a los allí declarados para un acceso posicional, la obtención de un nuevo tipo de iterator y la obtención de una "vista" representando una porción de la lista.

Los elementos en una lista conservan una secuencia, esta es el orden de introducción. Esta secuencia es la utilizada por los dos tipos de iterator a la hora de devolver los elementos en la lista.

Una lista permite elementos determinados equivalentes por equals, y la existencia de varios elementos nulos. Aunque también podría crease una versión especial que no lo hiciera.

Una lista podría contenerse a si misma, pero dado que equals y hashCode ya no están bien definidos en ella, debería emplearse con máxima precaución.

Nota sobre ordered y sorted: no son lo mismo, las listas tienen una secuencia independiente del valor de sus elementos; mientras un TreeSet esta sorted pues sus elementos están dispuestos dependiendo de su valor relativo (ordenados).

La primera posición en una lista es la posición cero.

Métodos con modificación de contrato

public boolean add(Object) (opcional)
Idéntico al definido en Collecion excepto que el argumento es añadido al final de la secuencia.public boolean equals(Object)
Devuelve verdadero si el argumento es una lista con el mismo tamaño y elementos equivalentes en la misma secuencia que la lista sobre la cual se invoca el método. Para los elementos que sean objetos la equivalencia viene determinada por equals. Si el elemento en cuestión es nulo en una lista, también ha de serlo en la otra.public int hashCode()
La expresión de calculo del valor hash de la lista se basa en valor de cada uno de los elementos en ella.

Ver la API. Se asegura, como especifica hashCode en Object, que dos listas produzcan el mismo valor si son determinadas equivalentes por equals.
public Iterator iterator()
Retorna un objeto que itera sobre los elementos de la lista en la secuencia propia de la lista.

La secuencia es el orden de introducción de los elementos, pero las utilidades de la clase Collection pueden modificarla.

```
import java.util.*;
//TestingIterator.java

public class TestingIterator {
  public static void main(String[] args) {
    List al = new ArrayList();
    for(int i=0; i<6; i++) al.add(new Integer(i));
    for(Iterator it= al.iterator() ; it.hasNext();)
            System.out.println(it.next());
    System.out.println("El iterator itera sobre los elementos" +
                    "según su orden de entrada");
    Collections.reverse(al);
    System.out.println(al);
    for(Iterator it= al.iterator() ; it.hasNext();)
            System.out.println(it.next());
    Collections.shuffle(al);
    System.out.println(al);
    for(Iterator it= al.iterator() ; it.hasNext();)
            System.out.println(it.next());
    System.out.println("Pero las modificaciones en la secuencia" +
```

```
                "como sort, reverse y shuffle son tenidas"+
                "en cuenta");
  }
}

/*
C:\Java\TIJ2\ch9>java  TestingIterator
0
1
2
3
4
5
El iterator itera sobre los elementos seg·n su orden de entrada
[5, 4, 3, 2, 1, 0]
5
4
3
2
1
0
[5, 1, 3, 0, 2, 4]
5
1
3
0
2
4
Pero las modificaciones en la secuencia como sort, reverse
y shuffle son tenidas en cuenta
*/
```

public boolean remove(Object) (opcional)
Idéntico al definido en Collection excepto que que elimina la primera ocurrencia (en secuencia) del argumento.

Métodos nuevos: acceso posicional

Pueden emplear un tiempo proporcional a la posición en algunas implementaciones, por ejemplo LinkedList. Por lo tanto, si se desconoce la implementación es preferible usar la iteración.

public void add(int, Object) (opcional)
Inserta el objeto en la posición indicada por int. El elemento en esa posición y todos los posteriores ven incrementados sus indices en uno.

Excepciones:

ClassCastException. Si el tipo del argumento no es el adecuado para la adición (infringe una restricción)

IllegalArgumentException. Si la restricción es cualquier otra no relacionada con el tipo del argumento.

UnsupportedOperationException. Si la implementación de este interfaz no incorporó este método.

IndexOutOfBoundsException. Si int es menor que cero o mayor que el tamaño de la lista.
public Object remove(int) (opcional)
Elimina de la lista el objeto en posición int y lo devuelve. Resta uno a los elementos situados tras él.

Excepciones:

UnsupportedOperationException. Si la implementación de este interfaz no incorporó este método.

IndexOutOfBoundsException. Si int e menor que cero o mayor que el tamaño de la lista.
public Object get(int)
Devuelve el elemento de indice especificado.

Lanza IndexOutOfBoundsException si int es menor que cero o mayor que el tamaño de la lista.
public Object set(int, Object) (opcional)
Remplaza y devuelve el elemento de posición int por su argumento.

Excepciones:

ClassCastException. Si el tipo del argumento no es el adecuado para la adición (infringe una restricción)

IllegalArgumentException. Si la restricción es cualquier otra no relacionada con el tipo del argumento.

UnsupportedOperationException. Si la implementación de este interfaz no incorporó este método.

IndexOutOfBoundsException. Si int es menor que cero o mayor que el tamaño de la lista.
public boolean addAll(int, Collection) (opcional)
Añade todos los elementos de la colección especificada en la posición int.

Devuelve verdadero si la lista se modifico como consecuencia de esta llamada.

Excepciones:

ClassCastException. Si el tipo del argumento no es el adecuado para la adición (infringe una restricción)

IllegalArgumentException. Si la restricción es cualquier otra no relacionada con el tipo del argumento.

UnsupportedOperationException. Si la implementación de este interfaz no incorporó este método.

IndexOutOfBoundsException. Si int es menor que cero o mayor que el tamaño de la lista.

Métodos nuevos: búsqueda secuencial

Pueden emplear un tiempo lineal (caro) en algunas implementaciones.

public int indexOf(Object)
Devuelve la posición de la primera ocurrencia del argumento. Si este es distinto de nulo equals determina la equivalencia, si es nulo se busca un elemento nulo con “==”. Devuelve -1 al no encontrarlo.public int lastIndexOf(Object)
Devuelve la última ocurrencia del argumento. Si este es distinto de nulo equals determina la equivalencia, si es nulo se busca un elemento nulo con “==”. Devuelve -1 al no encontrarlo.

Métodos nuevos: listIterator y subList

public ListIterator listIterator()
Devuelve un ListIterator cuyo puntero está situado antes del primer (en secuencia) elemento en la lista.public ListIterator listIterator(int)
Devuelve un ListIterator cuyo puntero está situado antes del elemento de indice int. Este método puede ser utilizado para retomar una iteración en el punto abandonado, puesto que la primera llamada a next sobre este ListIterator produce el elemento de posición int; a diferencia del método anterior que retornaría el primer elemento de la secuencia.

Lanza IndexOutOfBoundsException si int es menor que cero o mayor que el tamaño de la lista.
public List subList(intFrom, intTo)
Devuelve una vista de la lista formada por los elementos situados desde la posición dada por el primer argumento incluido, hasta el elemento anterior al situado en la posición dada por el segundo argumento.

La lista devuelta es respaldada por una porción de la lista sobre la cual se invoco el método. Las modificaciones a un elemento de cualquiera de ellas se observan en el correspondiente elemento de la otra.

Un modificación estructural es aquella que varia el tamaño de la lista o es capaz de perturbar una iteración en curso.

Pueden realizarse con seguridad modificaciones estructurales sobre la sublista. Pero si se realizan sobre la lista que la respalda, el comportamiento de la sublista no esta especificado.

```
import java.util.*;
//TestSubList.java

public class TestSubList {
  public static void main(String[] args) {
    ArrayList al = new ArrayList();
    for(int i=0; i< 10; i++) al.add(new MyInteger(i));
    List sub = al.subList(5, 10);
    System.out.println(sub.getClass());
    System.out.println("lista: " + al + "\nsublista: " + sub);
    System.out.println("Al modificar un elemento en una lista" +
                       "se observa en la"otra \nAdemas la sublista"+
                       "cuenta los indices desde cero, no desde la"+
                       "lista que la respalda");
    ((MyInteger) al.get(8)).i = 20;
    ((MyInteger) sub.get(0)).i = 20;
    System.out.println(al + "\n" + sub);
    System.out.println("Podemos hacer cualquier modificacion" +
                       "estructual a la sublista y se vera" +
                       "reflejado en la lista");
    sub.remove(0);
    System.out.println(sub + "\n" + al);
    Collections.sort(sub, Collections.reverseOrder());
    System.out.println(sub + "\n" + al);
    sub.add(new MyInteger(5));
    System.out.println(sub + "\n" + al);
    System.out.println("Pero basta que modifiquemos estructuralmente"+
                       "la lista para que cualquier acceso a la"+
                       "sublista produzca\n" +
                                  "ConcurrentModificationException");
    al.remove(0);
    System.out.println(al);
    System.out.println(sub);//ConcurrentModificationException
    sub.remove(0);//ConcurrentModificationException
    sub.add(new MyInteger(21));//ConcurrentModificationException
    sub.lastIndexOf(new MyInteger(5));//ConcurrentModificationException
    sub.indexOf(new MyInteger(5));//ConcurrentModificationException
  }

  static class MyInteger implements Comparable {
    int i;
    MyInteger(int i) { this.i=i; }
    public String toString() { return i+""; }
    public boolean equals(Object o) {
            if(o==null) return false;
            return ((MyInteger) o).i == i;
    }
    public int compareTo(Object o) {
            int oi = ((MyInteger) o).i;
```

```
            return i< oi ? -1 : ( i==oi ? 0 : 1);
    }
  }
}

/*
C:\Java\TIJ2\ch9>java TestSubList
class java.util.SubList
lista: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sublista: [5, 6, 7, 8, 9]
Al modificar un elemento en una lista se observa en la otra
Ademas la sublista cuenta los indices desde cero, no desde
la lista que la respalda
[0, 1, 2, 3, 4, 20, 6, 7, 20, 9]
[20, 6, 7, 20, 9]
Podemos hacer cualquier modificacion estructual a la sublista
y se vera reflejado en la lista
[6, 7, 20, 9]
[0, 1, 2, 3, 4, 6, 7, 20, 9]
[20, 9, 7, 6]
[0, 1, 2, 3, 4, 20, 9, 7, 6]
[20, 9, 7, 6, 5]
[0, 1, 2, 3, 4, 20, 9, 7, 6, 5]
Pero basta que modifiquemos estructuralmente la lista
para que cualquier acceso a la sublista produzca
ConcurrentModificationException
[1, 2, 3, 4, 20, 9, 7, 6, 5]
Exception in thread "main" java.util.ConcurrentModificationException
 at java.util.SubList.checkForComodification(Unknown Source)
 at java.util.SubList.listIterator(Unknown Source)
 at java.util.AbstractList.listIterator(Unknown Source)
 at java.util.SubList.iterator(Unknown Source)
 at java.util.AbstractCollection.toString(Unknown Source)
 at java.lang.String.valueOf(Unknown Source)
 at java.io.PrintStream.print(Unknown Source)
 at java.io.PrintStream.println(Unknown Source)
 at TestSubList.main(TestSubList.java:29)

En este sentido el comportamiento de la sublista devuelta
por una ArrayList es parecido al fallo-rapido de un Iterator.
También lo es el de la sublista devuelta por una LinkedList
*/
```

Lanza IndexOutOfBoundsException si intFrom es menor que cero o mayor que intTo, o intTo es mayor que el tamaño de la lista.

9.Interfaz ListIterator

Este tipo especial de Iterator (deriva de él) puede obtenerse de un List, no de un Set. También es fallo-rápido.

Añade métodos para iterar hacia atrás, para insertar o remplazar durante la iteración y para obtener la posición del puntero interno.

Entre cada par de elementos, y también antes del primero y después del último existe una posible posición del puntero:

(0) ele0 (1) ele1 (2) ele2 (3) ele3 (4)

El primer elemento tiene un indice cero, mientras que el último indice no corresponde a ningún elemento.

La primera posición posible del puntero es cero y la última el tamaño de la lista.

El método ListIterator() de una lista devuelve un ListIterator cuyo puntero es cero. El método ListIterator(int) posiciona el puntero en el indice int.

```
import java.util.*;
//TestListIterator.java

public class TestListIterator {
  public static void main(String[] args) {
    List al = new ArrayList();
    for(int i=0; i< 6; i++)  al.add(new Integer(i));
    ////////////////////next, previous, nextIndex y previousIndex
    ListIterator lit = al.listIterator();
    print("al.listIterator()...");
    print("posicion anterior del puntero " + lit.previousIndex());
    print("posicion del puntero " + lit.nextIndex());
    print("retornando el elemento actual " + lit.next());
    print("incrementa la posicion del puntero " + lit.nextIndex());
    print("retornando el elemento anterior " + lit.previous());
    print("decrementa el puntero " + lit.nextIndex());
    try { lit.previous(); }
    catch(NoSuchElementException e) {
            print("al comienzo de la lista previous() produce " + e);
    }
    lit = al.listIterator(al.size());
    print("al.listIterator(al.size())...");
    print("posicion actual " + lit.nextIndex());
    print("indice anterior " + lit.previousIndex());
    try{ lit.next(); }
    catch(NoSuchElementException e) {
            print("al final de la lista next() produce " + e);
    }
    print("retornando el elemento anterior decrementa el puntero"+
          lit.previous());
    print("posicion actual " + lit.nextIndex());
    print("retornando el elemento actual incrementa el puntero " +
         lit.next());
    print("posicion actual " + lit.nextIndex());
    ////////////////Iterando y buscando un elemento y su indice
    Integer missing = new Integer(3);
    print("buscando el elemento 3 desde el principio hacia"+
         "el final...");
    for(lit = al.listIterator(); lit.hasNext();) {
            if(lit.next().equals(missing))
              print("el indice viene dado por previousIndex() "+
            lit.previousIndex());
    }
    print("buscando el elemento 3 desde el final hacia"+
         el principio...");
    for(lit = al.listIterator(al.size()); lit.hasPrevious();) {
            if(lit.previous().equals(missing))
              print("el indice viene dado por nextIndex() " +
        lit.nextIndex());
    }
    ///////////////remplazando
    lit = al.listIterator(2);
    print("es necesario invocar next o previous antes de set...");
    try { lit.set(new Integer(-2)); }
    catch(IllegalStateException e)
```

```
            { print("o obtendremos " + e); }
    print("remplazando el elemento 2 con set tras haber"+
        "llamado a next");
    lit.next();
    lit.set(new Integer(-2));
    print(al);
    print("remplazando el elemento 2 de nuevo sin haber llamado"+
        "a next otra vez");
    lit.set(new Integer(-3));
    print(al);
    print("sin embargo si llamamos a add o remove e intentamos"+
        "otro set sin next o remove...");
    lit.remove();
    try { lit.set(new Integer(-4)); }
    catch(IllegalStateException e)
            { print("obtenemos " + e); }
    print(al);
    ///////////////insertando
    print("posicion actual " + lit.nextIndex());
    print("No es necesario invocar next o previous antes de add...");
    lit.add(new Integer(2));
    print(al);
    print("tras una inserccion el puntero y todos los indices"+
        "de los elementos");
    print("posteriores son incrementados en uno y por lo tanto"+
        "nextIndex devuelve el indice\nque devolveria antes de"+
        "la llamada mas uno");
    print("posicion actual " + lit.nextIndex());
    print("next retorna el mismo elemento que antes de la inserccion"+
        lit.next());
    lit.previous();//necesario porque lit.next incremento el puntero
    print("pero el elemento retornado por previous ahora es el recien"+
        "insertado "+lit.previous() );
    print("podemos inserta nuevamente sin llamar a next o previous");
    lit.add(new Integer(-2));
    print(al);
    //////////////eliminando
    print("posicion actual " + lit.nextIndex());
    print("intentar remove cuando el anterior fue add, es decir sin"+
        "next o previous");
    try { lit.remove(); }
    catch(IllegalStateException e)
      { print("produce " + e); }
    print("previous permite que remove elimine el elemento anterior a la "+
        "posicion actual " + lit.previous());
    lit.remove();
    print(al);
    print("remove decrementa los indices de los elementos posteriores...");
    print("posicion actual " + lit.nextIndex());
    print("intentar remove cuando el anterior fue remove, es decir sin"+
        "next o previous");
    try { lit.remove(); }
    catch(IllegalStateException e)
            { print("produce " + e); }
    print("next permite que remove elimine el elemento actual " +
        lit.next());
    lit.remove();
    print(al);
  }
```

```
  static void print(Object o) { System.out.println(o); }
}

/*
C:\Java\TIJ2\ch9>java TestListIterator
al.listIterator()...
posicion anterior del puntero -1
posicion del puntero 0
retornando el elemento actual 0
incrementa la posicion del puntero 1
retornando el elemento anterior 0
decrementa el puntero 0
al comienzo de la lista previous() produce
java.util.NoSuchElementException
al.listIterator(al.size())...
posicion actual 6
indice anterior 5
al final de la lista next() produce
java.util.NoSuchElementException
retornando el elemento anterior decrementa el puntero 5
posicion actual 5
retornando el elemento actual incrementa el puntero 5
posicion actual 6
buscando el elemento 3 desde el principio hacia el final...
el indice viene dado por previousIndex() 3
buscando el elemento 3 desde el final hacia el principio...
el indice viene dado por nextIndex() 3
es necesario invocar next o previous antes de set...
o obtendremos java.lang.IllegalStateException
remplazando el elemento 2 con set tras haber llamado a next
[0, 1, -2, 3, 4, 5]
remplazando el elemento 2 de nuevo sin haber llamado a next
otra vez
[0, 1, -3, 3, 4, 5]
sin embargo si llamamos a add o remove e intentamos otro set
sin next o remove...
obtenemos java.lang.IllegalStateException
[0, 1, 3, 4, 5]
posicion actual 2
No es necesario invocar next o previous antes de add...
[0, 1, 2, 3, 4, 5]
tras una inserccion el puntero y todos los indices de los
elementos posteriores son incrementados en uno y por lo tanto
nextIndexdevuelve el indice que devolveria antes de la llamada
mas uno
posicion actual 3
next retorna el mismo elemento que antes de la inserccion 3
pero el elemento retornado por previous ahora es el recien insertado 2
podemos inserta nuevamente sin llamar a next o previous
[0, 1, -2, 2, 3, 4, 5]
posicion actual 3
intentar remove cuando el anterior fue add, es decir
sin next o previous produce java.lang.IllegalStateException
previous permite que remove elimine el elemento anterior a
la posicion actual -2
[0, 1, 2, 3, 4, 5]
remove decrementa los indices de los elementos posteriores...
posicion actual 2
intentar remove cuando el anterior fue remove, es decir sin
next o previous produce java.lang.IllegalStateException
```

next permite que remove elimine el elemento actual 2
[0, 1, 3, 4, 5]
*/

public boolean hasNext()
Devuelve verdadero si el puntero es distinto del tamaño de la lista.public boolean hasPrevious()
Devuelve verdadero si el puntero es distinto de cero.public Object next()
Devuelve el elemento en cuyo indice se halla el puntero y avanza una posición el valor del mismo.

La primera vez que se invoca sobre el objeto retornado por el método ListIterator() devuelve el primer elemento de la lista. Cuando se invoca sobre el objeto retornado por ListIterator(int) devuelve el elemento de indice int. Si int fuera el tamaño de la lista lanzaría la excepción NoSuchElementException.

Lanza la excepción NoSuchElementException si la iteración ha alcanzado el final de la lista.
public Object previous()
Devuelve el elemento situado inmediatamente antes de la posición actual del puntero y resta uno a su valor.

Cuando es invocado sobre el objeto devuelto por ListIterator() lanza NoSuchElementException.

Si se llama sobre el objeto devuelto por ListIterator(int) devuelve el objeto situado en el indice int-1.

Lanza NoSuchMethodException si la iteración ha alcanzado el principio de la lista.
public int nextIndex()
Devuelve el indice del elemento que sería retornado por la próxima llamada a next, es decir la posición actual del puntero. Si el puntero se encuentra al final de la colección devuelve su tamaño.public int previousIndex()
Devuelve el indice del elemento que sería retornado por la próxima llamada a previous, es decir la posición actual del puntero menos uno. Devuelve -1 si el puntero se encuentra al comienzo de la lista.public void remove(Object) (opcional)
Elimina de la lista el último elemento retornado por next o previous. Solo puede ser llamado una vez por cada llamada a next o previous, y solo si no se invocó add después. Los indices de los elementos posteriores son decrementados en uno.

Excepciones:

UnsupportedOperationException. Si la implementación de este interfaz no incorporó este método.

IllegalStateException. Si next o previous no fueron llamados, o bien se invocó add o remove tras la última llamada a next o previous.
public void add(Object) (opcional)
Inserta el objeto en la lista en la posición actual del puntero y aumenta en uno su valor: la siguiente llamada a next quedaría inafectada, pero previous devolvería el elemento recién insertado. Los valores de los indices de elementos posteriores son incrementados en uno.

No es necesario haber invocado a next o previous con anterioridad.

Excepciones:

UnsupportedOperationException. Si la implementación de este interfaz no incorporó este método.

ClassCastException. Si el tipo de este objeto impidió su adición a un Set.

IllegalStateException. Si este objeto infringe alguna restricción al ser añadido a una Collection.
public void set(Object) (opcional)
Remplaza el último elemento producido por next o previous por el objeto especificado. Puede ser invocado varias veces sin necesidad de llamar nuevamente a next o previous, siempre y cuando no aparezcan add o remove entre dichas llamadas.

Excepciones:

UnsupportedOperationException. Si la implementación de este interfaz no incorporó este método.

ClassCastException. Si el tipo de este objeto impidió su adición a una lista.

IllegalStateException. Si este objeto infringe alguna restricción que impida su adicción a la lista. O bien, next o previous no fueron llamados, o lo fueron pero después de la invocación a add o remove.

10.Interfaz SortedSet

Deriva del interfaz Set. Garantiza que el Iterator obtenido de la clase que lo implemente recorrerá en sentido ascendente los elementos.

Aunque el interfaz no puede imponer la siguiente premisa la implementación de este interfaz proporcionada por la API, la clase TreeSet, si la sigue:

Toda implementación del interfaz debe proveer al menos cuatro constructores:

a) Un constructor sin argumentos crea un TreeSet vacío cuyos elementos deberán poseer un orden natural.

b) Constructor con un argumento Comparator que evite la necesidad de los elementos de contar con orden natural.

c) Un constructor con un argumento Collection creará un TreeSet conteniendo los elementos del argumento, los cuales deben contar con un orden natural.

d) Un constructor con un argumento Collection y otro Comparator creará un TreeSet con los elementos del primer argumento en el orden especificado por el segundo.

Todos los elementos han de ser mutuamente comparables, es decir, ni compareTo, ni compare deben arrojar ClassCastException para una pareja cualquiera de elementos.

El orden impuesto por los interfaces Comparable o Comparator debe ser consistente con equals. Si no lo fuera, el objeto TreeSet seguiría funcionando perfectamente, sin embargo no implementaría correctamente el interfaz Set. El orden impuesto por Comparable o Comparator es consistente con equals cuando se cumpla que si el método compareTo o compare determina que dos elementos son ordenados igualmente, equals establezca que ambos son eqivalentes. Esto es así porque si bien Set esta definido en función de equals, SortedSet esta definido en función de compareTo o compare. Por ejemplo, supongamos que un TreeSet posee un orden natural no consistente con equals.

Si el TreeSet rechaza la inclusión de un elemento porque el método compareTo determina que tiene idéntico orden a otro elemento ya existente en la colección, el mismo objeto TreeSet moldeado al tipo Set podría incluir dicho elemento porque equals no lo reconoce como duplicado.

```
import java.util.*;
//TestSortedSet.java

public class TestSortedSet {
  public static void main(String[] args) {
    SortedSet ss = new TreeSet();
    //un SortedSet siempre devuelve sus elementos en el orden natural
    //o establecido por Comparator. De hecho Collections.suffle o sort
    //no son aplicables a un Set, solo a un List
    for(int i=9; i>=0 ; i--) ss.add(new Integer(i));
            print(ss);
            //Los cambios realizados a una vista son observables en el
      //SortedSet y viceversa
            SortedSet view = ss.tailSet(new Integer(5));
            print(view);
            view.remove(new Integer(6));
            print(view + "\n" + ss);
```

```
            view.add(new Integer(6));
            print(view + "\n" + ss);
            //La inserción a una vista de un elemento fuera del rango con
      //el que fue creada provoca una excepción. La eliminación del
      //mismo elemento no provoca una excepcion puesto que no existe
      //en la vista.
            try { view.add(new Integer(4)); }
            catch(IllegalArgumentException e) { print(e); }
            view.remove(new Integer(4));
            print(view + "\n" + ss);
            //Podemos crear una vista de otra pero siempre que los argumentos
      //de la segunda se encuentren en el rango d ela primera. si no es
      //así obtenemos una excepción
            SortedSet viewOfView = view.headSet(new Integer(7));
            print(viewOfView);
            try { viewOfView = view.subSet(new Integer(4), new Integer(7)); }
            catch(IllegalArgumentException e) { print(e); }
    }
    static void print(Object o) { System.out.println(o); }
}

/*
C:\Java\TIJ2\ch9>java  TestSortedSet
//un SortedSet siempre devuelve sus elementos en el orden natural o
//establecido por Comparator. De hecho Collections.suffle o sort no son
//aplicables a un Set, solo a un List
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
//Los cambios realizados a una vista son observables en el SortedSet y
//viceversa
[5, 6, 7, 8, 9]
[5, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 7, 8, 9]
[5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
//La inserción a una vista de un elemento fuera del rango con el que fue
//creada provoca una excepción. La eliminación del mismo elemento no
//provoca una excepcion puesto que no existe en la vista.
java.lang.IllegalArgumentException: key out of range
[5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
//Podemos crear una vista de otra pero siempre que los argumentos de la
//segunda se encuentren en el rango de la primera. Si no es así obtenemos
//una excepción
[5, 6]
java.lang.IllegalArgumentException: fromKey out of range
*/
```

public Comparator comparator()
Devuelve el objeto Comparator asociado, o nulo si se emplea orden natural.public object first()
   Devuelve el primer (menor) elemento de la colección.

   Lanza NoSuchElementException si la colección esta vacía.
public object last()
   Devuelve el último (mayor) elemento de la colección.

   Lanza NoSuchElementException si la colección esta vacía.
public SortedSet subSet(fromObject, toObject)
   Devuelve una vista de este SortedSet respaldada por los elementos existentes desde fromObject incluido, hasta pero sin toObject. Las modificaciones realizadas en el SortedSet devuelto son visibles en el que recibió este mensaje y viceversa. Si los dos argumentos son iguales el tamaño del SortedSet

retornado es cero. El SortedSet devuelto soporta las mismas operaciones opcionales que el SortedSet que recibió este mensaje soporte.

Excepciones:

ClassCastException. Si los argumentos no pueden compararse entre sí mediante compareTo o compare.

Las implementaciones pueden, pero no es necesario, lanzar esta excepción si los argumentos no pueden compararse a los elementos contenidos en el SortedSet.

NullPointerException. Si alguno de los argumentos es null y la implementación de SortedSet no tolera elementos nulos.

IllegalArgumentException:

a) Si fromObject resulta ser mayor que toObject.

b) Si el SortedSet que recibe este mensaje resulta ser un SortedSet devuelto por los métodos subSet, headSet, o tailSet y alguno de los argumentos esta fuera del rango de los elementos con el que se creo.

c)Si se intenta insertar un elemento fuera del rango con el que se creo este SortedSet.
public SortedSet headSet(toObject)
Devuelve una vista de los elementos contenidos en este Sortedset que son menores que toObject. Los cambios realizados en el SortedSet son visibles en el que recibió este mensaje y viceversa.

El SortedSet devuelto soporta las mismas operaciones opcionales que el SortedSet que recibió este mensaje soporte.

Excepciones:

ClassCastException. Si el argumento no es compatible con los métodos compareTo o compare.

Las implementaciones pueden, pero no es necesario, lanzar esta excepción si el argumento no puede compararse a los elementos contenidos en el SortedSet.

NullPointerException. Si el argumento es nulo y la implementación de SortedSet no tolera elementos nulos.

IllegalArgumentException:

a) Si el SortedSet que recibe este mensaje resulta ser un SortedSet devuelto por los métodos subSet, headSet, o tailSet y el argumento esta fuera del rango de los elementos con el que se creo.

b) Si se intenta insertar un elemento fuera del rango con el que se creo este SortedSet.
public SortedSet taiSet(fromObject)
Devuelve una vista de los elementos contenidos en este Sortedset que son mayores o iguales a fromObject. Los cambios realizados en el SortedSet son visibles en el que recibió este mensaje y viceversa.

El SortedSet devuelto soporta las mismas operaciones opcionales que el SortedSet que recibió este mensaje soporte.

Excepciones:

ClassCastException. Si el argumento no es compatible con los métodos compareTo o compare.

Las implementaciones pueden, pero no es necesario, lanzar esta excepción si el argumento no puede compararse a los elementos contenidos en el SortedSet.

NullPointerException. Si el argumento es nulo y la implementación de SortedSet no tolera elementos nulos.

IllegalArgumentException:

a) Si el SortedSet que recibe este mensaje resulta ser un SortedSet devuelto por los métodos subSet, headSet, o tailSet y el argumento esta fuera del rango de los elementos con el que se creo.

b) Si se intenta insertar un elemento fuera del rango con el que se creo este SortedSet.

11.Interfaz Map

No deriva de Collection. Representa asociaciones entre parejas clave-valor. Las claves son únicas en el Map, los valores no tienen porque serlo. Una clave es asociada únicamente a un valor.

Puede obtenerse una colección del tipo Set conteniendo las claves del Map, otra de las parejas clave-valor, y otra del tipo Collection (implementada por una List) de los valores. El orden del Map es aquel establecido por estas colecciones. La implementación HashMap no garantiza ningún orden, mientras que TreeMap si lo hace.

Un Map no puede contenerse a sí mismo como clave, aunque podría hacerlo como valor, pero no es recomendable, puesto que equals y hashCode no serían correctamente implementados.

La operación del Map resulta impredecible si el estado de un objeto utilizado como clave es modificado, de forma que afecte a las comparaciones realizadas mediante equals, mientras se encuentra en el Map.

Aunque el interfaz no puede imponer la siguiente premisa, todas las implementaciones proporcionadas en el marco de trabajo si la cumplen:

Toda implementación debe contar con al menos dos constructores. Uno sin argumentos, y otro con un argumento del tipo Map. El primero crea un Map vacío, y el segundo conteniendo las parejas clave-valor que se encuentran en el Map pasado.

```
import java.util.*;
//TestMap.java

public class TestMap {
  public static void main(String[] args) {
    Map m = new HashMap();
    for(char c='a'; c <'f'; c++) m.put(new Integer((int) c),
       new Character(c));
    print(m);
    print("Intentar añadir una clave distinta a la contenida"+
         "en el Map produce");
    try { m.put(new Long(1), new Character('z')); }
    catch(ClassCastException e) { print(e); }
    print("Sin embargo podemos añadir cualquier valor en esta"+
         "implementacion");
    m.put(new Integer(1), new LinkedList());
    print(m + "\neliminado la LinkedList");
    m.remove(new Integer(1));
    print(m +"\nIntentar eliminar una pareja cuya clave no es"+
        "del tipo apropiado produce");
    try { m.remove(new LinkedList()); }
    catch(ClassCastException e) { print(e); }
    print("Tambien si intentamos 'get' con una clave de un tipo"+
         "no apropiado");
    try { m.get(new LinkedList()); }
    catch(ClassCastException e) { print(e); }
    print("contiene clave 97 "+ m.containsKey(new Integer((int) 'a')));
    print("Contiene valor a "+ m.containsValue(new Character('a')));
```

```
    print("containsKey produce error si la clave no es del tipo"+
        "apropiado");
    try { m.containsKey(new LinkedList()); }
    catch(ClassCastException e) { print(e); }
    print("pero no containsValue "+ m.containsValue(new LinkedList()));
    print("Vista de los valores " + m.values());
    print("Vista de las parejas " + m.entrySet());
    print("Vista de las claves " + m.keySet());
    print("Modificaciones en el Map y en la vista son visibles en ambos");
    Collection values = m.values();
    m.put(new Integer((int) 'f'), new Character('f'));
    values.remove(new Character('a'));
    print(values +"\n" + m);
    print("no podemos añadir a la vista...");
    try { values.add(new Character('a')); }
    catch(UnsupportedOperationException e) { print(e); }
    //el siguiente codigo entra en un bucle infito de
    //ConcurrentModificatioException lo que demuestra que el
    //comportamiento de la vista se vuelve indefinido si se modifica
    //el Map de forma distinta a Iterator.remove mientras se esta
    //iterando.
    /*boolean firstTime = true;
      for(Iterator it = values.iterator(); it.hasNext();) {
              try { print(it.next()); }
              catch(ConcurrentModificationException e) { print(e); }
              if(firstTime) {
                firstTime = false;
                m.put(new Integer((int) 'a'), new Character('a'));
              }
             }
    */
  }
  static void print(Object o) { System.out.println(o); }
}

/*
//Cuando m es un TreeMap:
C:\Java\TIJ2\ch9>java TestMap
{97=a, 98=b, 99=c, 100=d, 101=e}
Intentar a±adir una clave distinta a la contenida en el Map
produce java.lang.ClassCastException: java.lang.Integer
Sin embargo podemos a±adir cualquier valor en esta implementacion
{1=[], 97=a, 98=b, 99=c, 100=d, 101=e}
eliminado la LinkedList
{97=a, 98=b, 99=c, 100=d, 101=e}
Intentar eliminar una pareja cuya clave no es del tipo apropiado
produce java.lang.ClassCastException: java.util.LinkedList
Tambien si intentamos 'get' con una clave de un tipo no apropiado
java.lang.ClassCastException: java.util.LinkedList
contiene clave 97 true
Contiene valor a true
containsKey produce error si la clave no es del tipo apropiado
java.lang.ClassCastException: java.util.LinkedList
pero no containsValue false
Vista de los valores [a, b, c, d, e]
Vista de las parejas [97=a, 98=b, 99=c, 100=d, 101=e]
Vista de las claves [97, 98, 99, 100, 101]
Modificaciones en el Map y en la vista son visibles en ambos
[b, c, d, e, f]
{98=b, 99=c, 100=d, 101=e, 102=f}
```

```
no podemos a±adir a la vista...
java.lang.UnsupportedOperationException

//Cuando m es un HashMap:
C:\Java\TIJ2\ch9>java TestMap
{98=b, 97=a, 101=e, 100=d, 99=c}
Intentar a±adir una clave distinta a la contenida en el Map produce
Sin embargo podemos a±adir cualquier valor en esta implementacion
{98=b, 97=a, 101=e, 1=[], 1=z, 100=d, 99=c}
eliminado la LinkedList
{98=b, 97=a, 101=e, 1=z, 100=d, 99=c}
Intentar eliminar una pareja cuya clave no es del tipo apropiado produce
Tambien si intentamos 'get' con una clave de un tipo no apropiado
contiene clave 97 true
Contiene valor a true
containsKey produce error si la clave no es del tipo apropiado
pero no containsValue false
Vista de los valores [b, a, e, z, d, c]
Vista de las parejas [98=b, 97=a, 101=e, 1=z, 100=d, 99=c]
Vista de las claves [98, 97, 101, 1, 100, 99]
Modificaciones en el Map y en la vista son visibles en ambos
[b, f, e, z, d, c]
{98=b, 102=f, 101=e, 1=z, 100=d, 99=c}
no podemos a±adir a la vista...
java.lang.UnsupportedOperationException

//El comportamiento distinto se debe a que HashMap equals
//para los accesos al map.
equals no lanza ClassCastException si la clave no es del tipo
apropiado.
TreeMap emplea compareTo o compare que si lanza ClassCastException
si el tipo de la clave no es el apropiado.
*/
```

Manipulación individual

public Object put(keyObject, valueObject) (opcional)

Agrega la pareja clave-valor especificada al Map. Si la clave ya existe remplaza el valor existente por el proporcionado y lo devuelve. Si devuelve nulo puede indicar que no existía dicha clave o bien que estaba asociada a un valor nulo.

Excepciones:

UnsupportedOperationException. Si la implementación de este interfaz no incluye este método.

NullPointerException. Si alguno de los argumentos es nulo y esta implementación no soporta nulos.

ClassCastException. Si el tipo de algún argumento impide su inclusión en el Map.

IllegalArgumentException. Si algún otro aspecto de los argumento infringe una restricción de esta implementación.

public Object get(keyObject)

Devuelve el valor correspondiente a la clave pasada como argumento. Si retorna nulo la clave podría no estar contenida en el Map, o bien habría sido asociada con un valor nulo.

Excepciones:

NullPointerException. Si el argumento es nulo y esta implementación no soporta nulos.

ClassCastException. Si el tipo del argumento impide su búsqueda en el Map.
public Object remove(keyObject) (opcional)
Elimina del Map la pareja clave-valor cuya clave es la especificada y devuelve su valor. Si devuelve nulo puede indicar que la clave no estaba incluida en el Map, o que había sido asociada a nulo.

Excepciones:

UnsupportedOperationException. Si la implementación de este interfaz no incluye este método.

ClassCastException. Si el tipo del argumento impide su búsqueda en el Map.

Manipulación colectiva.

public void clear() (opcional)
Elimina todos las parejas contenidas en el Map.

Excepciones:

UnsupportedOperationException. Si la implementación de este interfaz no incluye este método.
public void putAll(Map) (opcional)
Añade al Map todas las parejas contenidas en Map pasado como argumento. Remplaza los valores de aquellas claves presentes en ambos Map.

Excepciones:

UnsupportedOperationException. Si la implementación de este interfaz no incluye este método.

NullPointerException. Si alguno de las claves o valores a añadir son nulos y esta implementación no soporta nulos.

ClassCastException. Si el tipo de algúna clave o valor impide su inclusión en el Map.

IllegalArgumentException. Si algún otro aspecto de las claves o valores infringe una restricción de esta implementación.

Obtención información.

public int size()
Devuelve el tamaño de este Map que igual al número de parejas contenidas.public boolean isEmpty()
Devuelve verdadero si el Map esta vacío.public boolean containsKey(Object)
Devuelve verdadero si el argumento es una clave contenida en el Map.

Excepciones:

NullPointerException. Si el argumento es nulo y esta implementación no soporta nulos.

ClassCastException. Si el tipo del argumento impide su búsqueda en el Map.
public boolean containsValue(Object)
Devuelve verdadero si el argumento es un valor contenido en el Map.

Obtención de vistas.

public Set entrySet()
Devuelve un Set conteniendo las parejas del Map. El tipo de estos objetos es “Map.Entry”. El Set es respaldado por el Map, luego una modificación en uno es visible en el otro. Una modificación en el Map mientras se esta iterando sobre el Set lo deja en un estado indefinido. El Set puede eliminar parejas del Map con cualquiera de sus métodos de supresión de elementos, pero no es posible añadir nuevas parejas.public Set keySet()

Devuelve un Set conteniendo las claves del Map. El Set retornado esta respaldado por el Map, luego los cambios en uno son visibles en el otro. La modificación del Map mientras se esta iterando sobre el Set retornado lo deja en un estado indefinido. Las parejas del Map pueden eliminarse con cualquier método de eliminación del Set, pero este no soporta la adicción de claves. (Añadir solo una clave no tiene sentido.)public Collection values()
Devuelve una Collection que no es un Set de los valores contenidos en el Map. La Collection retornada es respaldada por el Map, luego una modificación en uno es visible en el otro. La modificación del Map mientras se itera sobre el Collection lo deja en un estado indefinido. Las parejas del Map pueden eliminarse mediante cualquier método de eliminación de la Collection, pero esta no soporta la adición de valores. (Añadir únicamente un valor no tiene sentido.

equals y hashCode

Devuelve verdadero si el argumento es un Map con las mismas parejas que el Map sobre el que se invocó el método. Esto ocurre si equals determina que los Set obtenidos tras la invocación de entrySet sobre los Maps que se comparan son equivalentes.public int hashCode()
Devuelve el código hash de este Map. Es calculado como la suma de los códigos hash de las parejas en el Set devuelto por entrySet. De esta forma se asegura que si dos Maps son determinados equivalentes por equals posean el mismo código hash.

12.Interfaz Map.Entry

Es un interfaz anidado en el interfaz Map. Los objetos contenidos en el Set devuelto por entrySet son de este tipo.

La única forma de obtener una referencia a una pareja en el Map es a través del Iterator del Set mencionado. El comportamiento de dicha pareja (de tipo Map.Entry) es indefinido si el Map es modificado, después de la obtención de la pareja, mediante un método distinto a “Iterator.remove” o “Map.Entry.setValue”.

```
import java.util.*;
//TestMapEntry.java

public class TestMapEntry {
  public static void main(String[] args) {
    Map m = new TreeMap();
    for(char c ='a'; c < 'f';c++)  m.put(new Integer((int) c), new Character(c));
    Set entry = m.entrySet();
    print("map= " + m + "\n" + "entryset= " + entry);
    print("Iterando:");
    for(Iterator it = entry.iterator(); it.hasNext();) print(it.next());
    Iterator it = entry.iterator();
    print("La primera pareja e el map:");
    Map.Entry pareja = (Map.Entry) it.next();
    print(pareja.getKey() + " = " + pareja.getValue());
    print("Podemos modicar la entrada con setValue:");
    pareja.setValue(new Character('z'));
    print(pareja.getKey() + " = " + pareja.getValue());
    print("El cambio se refleja tanto en el map como en entrySet:");
    print("map= " + m + "\n" + "entrySet= " + entry);
    print("También podemos eliminar la pareja con Iterator.remove");
    it.remove();
    print("map= " + m + "\n" + "entrySet= " + entry);
    print("cualquier otra modificacion no invalida la pareja ya obtenida");
    print("(en desacuerdo con la API)");
    entry = m.entrySet();
    it = entry.iterator();
    pareja = (Map.Entry) it.next();
    print("pareja= " + pareja);
    m.put(new Integer((int) 'z'), new Character('z'));
```

```
    pareja.setValue(new Character('x'));
    print("map= " + m + "\n" + "entrySet= " + entry);
    print("en cambio si invalida el iterator ya obtenido (fallo-rapido)");
    try { pareja = (Map.Entry) it.next(); }
    catch(ConcurrentModificationException e) { print(e); }
    }
    static void print(Object o) { System.out.println(o); }
}

/*
C:\Java\TIJ2\ch9>java  TestMapEntry
map= {97=a, 98=b, 99=c, 100=d, 101=e}
entryset= [97=a, 98=b, 99=c, 100=d, 101=e]
Iterando:
97=a
98=b
99=c
100=d
101=e
La primera pareja e el map:
97 = a
Podemos modicar la entrada con setValue:
97 = z
El cambio se refleja tanto en el map como en entrySet:
map= {97=z, 98=b, 99=c, 100=d, 101=e}
entrySet= [97=z, 98=b, 99=c, 100=d, 101=e]
TambiÚn podemos eliminar la pareja con Iterator.remove
map= {98=b, 99=c, 100=d, 101=e}
entrySet= [98=b, 99=c, 100=d, 101=e]
cualquier otra modificacion no invalida la pareja ya obtenida
(en desacuerdo con la API)
pareja= 98=b //setValue todavía funciona:
map= {98=x, 99=c, 100=d, 101=e, 122=z}
entrySet= [98=x, 99=c, 100=d, 101=e, 122=z]
en cambio si invalida el iterator ya obtenido (fallo-rapido)
java.util.ConcurrentModificationException
*/
```

public Object getKey()
Devuelve la clave de esta pareja en el Map.public Object getValue()
Devuelve el valor de esta pareja en el Map. El comportamiento de este método no esta definido si la pareja fue eliminada del Map. (Mediante Iterator.remove.)public Object setValue() (opcional)

Remplaza el valor de esta pareja tanto en el Set devuelto por entrySet como en el Map sobre el que dicho método fue invocado. (Los cambios en uno son visibles en el otro. En realidad el 1º esta respaldado por el 2º.)

Devuelve el valor que fue remplazado.

Excepciones:

UnsupportedOperationException. Si la implementación del Map no incluye este método.

NullPointerException. Si el argumento es nulo y la implementación del Map no soporta nulos.

ClassCastException. Si el tipo del argumento impide su inclusión en el Map.

IllegalArgumentException. Si algún otro aspecto del argumento infringe una restricción de la implementación del Map.
public boolean equals(Object)

Devuelve verdadero si esta pareja clave-valor es equivalente a la pasada como argumento. Esto se cumple si ambas claves son equivalentes, según el método equals, o bien, las dos son nulas; y solo si ocurre lo mismo con los valores.public int hashCode()
Devuelve el código hash de esta pareja clave-valor. Es calculado como la o-exclusiva de los códigos hash de la clave y del valor. Si la clave o valor fueran nulo, se utiliza cero. De esta forma se asegura que si dos objetos Map.Entry son determinados equivalentes por equals, posean el mismo código hash.

13.Interfaz SortedMap

Deriva de Map. La claseTreeMap lo implementa. Asegura un orden en sus parejas clave-valor establecido "naturalmente" por las claves o mediante un Comparator. Al ser análogo a SortedSet no va ser tratado aquí.

Consulte la API y la descripción de SortedSet para los métodos declarados en este interfaz:

Comparator comparator()
Object firstKey()
Object lastKey()
SortedMap subMap(fromObject, toObject)
SortedMap headMap(toObject)
SortedMap tailMap(fromObject)

14.Factor de carga y capacidad

Los objetos de tipo Map y Collection son contenedores, que a diferencia de las matrices, incrementan su capacidad según la inclusión de nuevos elementos la agota.

La capacidad de uno de estos contenedores es el número de elementos que la estructura de datos actual podría llegar a almacenar. El tamaño (retornado por size) es el total de elementos contenidos en dicha estructura. El factor de carga es la relación tamaño/capacidad. Cuando el número de elementos contenidos hacen que se sobrepase dicha relación la capacidad de la estructura es incrementada: básicamente creando una estructura nueva y copiando los elementos de una a otra. Si la estructura de datos es una tabla hash, este proceso implica la distribución uniforme de los elementos en la nueva tabla. Si el objeto contenedor es un HashMap, el HashMap recién creado tiene una capacidad doble que el antiguo, y este proceso se realiza mediante la llamada a un método privado llamado rehash.

15.La clase ArrayList

Conserva una secuencia entre sus elementos, pero no un orden a menos que se utilice Collections.sort.

Deriva de la clase AbstractList e implementa List, Clonable y Serializable. Mantiene sus elementos en una matriz.

Los métodos size, get, isEmpty, iterator y listIterator se ejecutan en un tiempo constante menor el correspondiente a LinkedList. El método add se ejecuta en un tiempo constante amortizado, es decir, añadir n elementos conlleva un tiempo n multiplicado por el factor constante. Los métodos restantes se ejecutan en un tiempo lineal.

Sus iterators son fallo-rapido. Permite duplicados y nulos.

No esta sincronizado. Para procurar un acceso seguro por parte de varios hilos de ejecución puede sincronizarse sobre el objeto que la contenga o utilizar Collection.synchronizedList.

Dado que la operación de incremento de la capacidad de la matriz que contiene los elementos lleva su tiempo, ante la perspectiva de la inclusión de muchos elementos nuevos es conveniente invocar al método ensureCapacity para evitar ampliaciones sucesivas.

El acceso directo es eficiente, no así el secuencial. Mientras la adición o eliminación al comienzo o final es realizada en un tiempo aceptable, la implementación LinkedList la supera ampliamente cuando estas operaciones se realizan en posiciones interiores. Por lo general, es la implementación que por defecto

deberíamos escoger, y optar por LinkedList únicamente si deseamos añadir o eliminar con frecuencia en posiciones interiores.

Se muestran los métodos adicionales al interfaz List, así como un constructor no “especificado” por Colection.

public ArrayList(int)
Construye un objeto ArrayList cuya matriz tiene el tamaño especificado.

Lanza IllegalArgumentException si int es negativo.
public void ensureCapacity(minimoInt)
Si la capacidad actual de la matriz es inferior al argumento la incrementa, asegurando que la capacidad permitirá al menos dicho número de elementos.public void trimToSize()
Modifica la capacidad de la matriz para que sea igual al número de elementos contenidos. protected void removeRange(fromInt, toInt)
Elimina el elemento situado en el indice especificado por el primer argumento y todos los siguientes menores que el indice dado por el segundo. Decrementa los indices de los elementos posteriores al rango en uno.

16.La clase LinkedList

Conserva una secuencia entre sus elementos, pero no un orden a menos que se utilice Collections.sort.

Deriva de la clase AbstractSequentialList, la cual a su vez extiende de AbstractList. Implementa List, Cloneable y Serializable. Su estructura de datos es un lista enlazada. Las operaciones de acceso posicional empiezan a recorrer la lista desde el extremo mas próximo al indice. El recorrido secuencial y la inserción-extracción de posiciones intermedias son mas rápidas que en la implementación ArrayList.

Esta dotada de una serie de operaciones adicionales a las declaradas en el interfaz List. Estas operaciones (que también podrían ser realizados mediante métodos definidos en List) permiten utilizarla como un pila, cola y cola doblemente terminada (dequeue):

getFirst(), getLast(), removeFirst(), removeLast(), addFirst() y addLast().

Los cuatro primeros lanzan NoSuchElementException si la lista esta vacía.

17.La clase Vector

Es una implementación heredada desde JDK 1.1. En JDK 1.2 fue adaptada para que implementase List. De hecho deriva de AbstracList.

No se recomienda su uso. No es mas rápida que las implementaciones ArrayList o LinkedList, sin embargo a diferencia de ellas si esta sincronizada. Sus iterators también son fallo-rápido.

Estos son los métodos y constructores adicionales a List que vector posee:

public Vector(int)
El vector creado tiene la capacidad especificada. Cuando se utiliza el constructor sin argumentos la capacidad inicial es diez.public Vector(capacidadInt, incrementoInt)
El vector creado tiene la capacidad especificada. El segundo elemento es el número en que la capacidad será incrementada cuando el tamaño sea mayor que la capacidad.void addElement(Object)
void removeElement(Object)
Object elementAt(int)
Object firstElement()
Object lastElement()
setElementAt(Object, int)
void insertElementAt(Object, int)
int indexOf(Object, int)
int lastIndexOf(Object, int)
boolean removeElement(Object)
Object removeElementAt(int)

void removeAllElements()
protected void removeRange(fromInt, toInt)
void trimToSize()
void copyInto(Object[])
void ensureCapacity(int)
int capacity()
Enumeration elements()

18.La clase Stack

Deriva de Vector e implementa List, Cloneable y Serializable. También es una implementación heredada de JDK 1.1. En JDK 1.2 añade unos métodos a Vector que permiten tratarlo como una pila:

boolean empty()
Object push(Object)
int search(Object)
Object pop()
Object peek()

19.La clase HashSet

Deriva de AbstractSet e implementa Set, Serializable y Cloneable.

No permite duplicados pero si un único elemento nulo.

Su estructura de datos es un HashMap. Cuando un elemento nuevo pretende ser añadido se utiliza su valor hash para indexar en la estructura. Además el método equals del elemento es utilizado para determinar si el elemento nuevo es equivalente a los contenidos en el cubo "bucket" obtenido en el paso anterior.

Ofrece un funcionamiento en un tiempo constante para las operaciones básicas: add, remove, size y contains. Siempre y cuando la función hash disperse los elementos adecuadamente entre los cubos.

La iteración a través del HashSet es proporcional a la suma de su tamaño y su capacidad. Por lo tanto, si el tiempo de iteración es importante, no debe crearse el objeto con una capacidad elevada o un factor de carga pequeño.

Esta implementación no esta sincronizada por defecto, sino que suele utilizarse el objeto que la contiene, o bien Collecction.synchronizedSet. Su iterator es fallo-rápido.

Siguen los dos constructores no "especificados" en el interfaz Collection:

public HashSet(int)
Crea un objeto de la capacidad especificada.

Lanza IllegalArgumentException si int es negativo.
public HashSet(capacidadInt, factorFloat)
Crea un objeto de la capacidad y factor de carga especificados.

El constructor sin argumentos establece un factor de carga 0.75.

Lanza IllegalArgumentException si uno de los argumentos fuera negativo.

20.La clase TreeSet

Deriva de AbstractSet e implementa SortedSet, Cloneable y Serializable. No permite duplicados pero si un único elemento nulo.

Es respaldado por un TreeMap. Asegura un tiempo de ejecución log(n) para las operaciones básicas: add, remove, y size.

Mantiene un orden ascendente si los elementos cuentan con ordenación natural. O bien mantiene el orden impuesto por un Comparator. Este orden ha de ser consistente con equals. Si no lo fuera, el TreeSet sigue siendo totalmente funcional pero ya no implementaría correctamente el interfaz Set. El interfaz Set esta definido en base a equals, pero un objeto TreeSet realiza todos los accesos aplicando compareTo o compare al elemento a acceder o añadir.

Ambos métodos deben lanzar ClassCastException si el elemento que se pretende agregar al Set no es del tipo adecuado.

El iterator de esta implementación es fallo-rápido. Esta implementación no esta sincronizada. Si se requiere, puede sincronizarse sobre el objeto que contiene al TreeSet, o bien usando Collection.syunchronizedSet.

Se muestran los constructores que no fueron "especificados" por el interfaz Collection:

public TreeSet(TreeSet)
public TreeSet(Comparator)

21.La clase HashMap

Deriva de AbstractMap e implementa Map, Serializable y Cloneable.

Asocia un objeto clave a otro objeto denominado valor.

No permite duplicados en la clave pero si en el valor. Tanto uno como el otro pueden ser nulos.

Es una implementación basada en un tabla hash. Cuando una pareja calve-valor nueva pretende ser añadida se utiliza el valor hash de la clave para indexar en la tabla hash. Además el método equals de la clave es utilizado para determinar si el elemento nuevo es equivalente a los contenidos en el cubo "bucket" obtenido en el paso anterior.

No se mantiene ningún orden entre las parejas contenidas.

Ofrece un funcionamiento en un tiempo constante para las operaciones básicas: put y get. Siempre y cuando la función hash disperse las parejas adecuadamente entre los cubos.

La iteración a través del HashMap es proporcional a la suma de su tamaño y su capacidad. Por lo tanto, si el tiempo de iteración es importante, no debe crearse el objeto con una capacidad elevada o un factor de carga pequeño. Si se esperan muchas adiciones a un HashMap crearlo con una capacidad adecuada evitará incrementos sucesivos en su capcidad,

Esta implementación no esta sincronizada por defecto, sino que suele utilizarse el objeto que la contiene, o bien Collecction.synchronizedMap. Todos los iterators obtenidos de las vistas del HashMap son fallo-rápido.

Siguen los dos constructores no "especificados" en el interfaz Map:

public HashMap(int)
  Crea un objeto HashMap de capacidad especificada y un factor de carga 0.75

  Lanza IllegalArgumentException si int es negativo.
public HashMap(capacidadInt, factorFloat)
  Crea un objeto HashMap de la capacidad y factor de carga especificados.

  Lanza IllegalArgumentException si algún argumento es negativo.

22.La clase TreeMap

Deriva de AbstractMap e implementa SortedMap, Cloneable y Serializable. Al ser un Map también asocia parejas clave-valor.

Es respaldado por un árbol Rojo-Negro. Asegura un tiempo de ejecución log(n) para las operaciones : put, get, remove y containsKey.

Mantiene un orden ascendente si las claves cuentan con ordenación natural. O bien mantiene el orden impuesto por un Comparator. Este orden ha de ser consistente con equals. Si no lo fuera, el TreeMap sigue siendo totalmente funcional pero ya no implementaría correctamente el interfaz Map. El interfaz Map esta definido en base a equals, pero un objeto TreeMap realiza todos los accesos aplicando compareTo o compare a la clave a acceder o añadir.

Ambos métodos deben lanzar ClassCastException si la clave que se pretende agregar al Set no es del tipo adecuado.

Los Iterators que se obtienen de las vistas de esta implementación son todos fallo-rápido. Esta implementación no esta sincronizada. Si se requiere, puede sincronizarse sobre el objeto que contiene al Treemap, o bien usando Collection.syunchronizedMap.

Se muestran los constructores que no fueron “especificados” por el interfaz Map:

public TreeMap(TreeMap)>/code>
public TreeMap(Comparator)

23.La clase WeakHashMap

Mantiene sus claves a través de WeakObjects. Sus valores son apuntados por referencias normales. Cuando el recolector de basuras encuentra objetos claves referidos únicamente por referencias débiles, es decir cuando las claves ya no son apuntadas por otras variables desde fuera del map, el recolector borra dichas referencias y los objetos clave pueden ser reciclados. La próxima vez se acceda al map este eliminará las entradas correspondientes a dichas claves.

Por lo tanto este map se comporta como si un hilo estuviera eliminando parejas clave-valor. Por este motivo sucesivas llamadas a sus métodos podrían retornar valores inconsistentes.

Esta implementación esta dirigida principalmente al uso con claves cuyo equals esta basado en ==, en vez de comparaciones sobre el estado de los objetos. Los objetos que implementan equals con comparaciones del estado de los objetos son objetos recreables. Aquellos en los que equals se basa en == no lo son. Si utilizamos objetos no recreables como claves no sería posible sorprenderse si al realizar una búsqueda de un objeto que ya fue desechado ya no lo encontrásemos en el map: el contenedor lo habría eliminado y al volver a crearlo no podríamos esperar que equals determinase equivalencia con ningún otro objeto.

Por lo demás WeakHashMap es equivalente a HashMap.

24.La clase HashTable

Deriva de java.util.Dictionary e implementa Map, Cloneable y Serializable. Dictionary es una clase abstracta y obsoleta. De hecho HashTable es otra implementación heredada desde JDK 1.1 que se mantiene por criterios de compatibildad. HashMap o TreeeMap deberían de ser elegidos preferentemente.

No permite claves o valores nulos. Los iterators obtenidos de sus vistas son fallo-rapido. Como el resto de las implementaciones heredadas esta sincronizado.

Estos son los métodos y constructores no “declarados” en el interfaz Map:

public HasHTable(int)
Construye un HashTable con la capacidad especificada y un factor de carga 0.75public HashTable(capacityInt, factorInt)
Construye un HashTable con la capacidad y factor de carga especificados.public boolean contains(Object)
Devuelve verdadero si el HashTable contiene la clave especificada. Realiza una función similar a Map.containsKey pero es menos eficiente.public Enumeration elements()
Devuelve un objeto Enumeration conteniendo los valores en el HashTable.public Enumeration keys()
Devuelve un objeto Enumeration conteniendo las claves en el HashTable.protected void rehash()

Este método es llamado automáticamente cuando cuando el número de elementos hace que se exceda la relación del factor de carga.public String toString()
Devuelve una representación imprimible del contenido del HashTable.

25.Sobrescribiendo los métodos hashCode y equals del objeto contenido

Una clase que vaya a ser utilizada como clave en un HashMap o Hashtable, o bien como elemento en un HashSet, ha de proveer un implementación adecuada de estos métodos.

Aplicando hashCode sobre la clave o elemento devuelve un entero que será utilizado para indexar en la estructura hash. De esta forma se obtiene rápidamente la posición que ocupa la clave o elemento. Como la tabla puede ser menor que el número de objetos a guardar no es requerido que el valor devuelto por hashCode sea único para cada objeto. Luego en una posición de la estructura pueden estar almacenados mas de un elemento (en una ArrayList por ejemplo), para saber si la clave ya esta almacenada en la tabla se invoca a equals que deberá determinar si la clave con la se accede es equivalente a alguna de las claves residentes en la posición indexada por el código hash de la clave.

equals devolverá verdadero si el objeto que recibe como argumento es del mismo tipo, no nulo y es considerado idéntico al cual recibió este mensaje. La comparación debe establecerse entre los campos de los objetos. Los campos a considerar son determinados por el programador en base a la naturaleza del objeto.

La implementación de hashCode debe contar con las siguientes características :

Retornará siempre el mismo valor cuando es invocado sobre el mismo objeto.

Debe ser significativo. Esto es, ha de calcularse considerando todos los campos de un objeto que lo distinguen de otras instancias de la misma clase.

No tiene por que único para un objeto. En particular “this” es un valor pésimo como código hash pues es imposible crear otro objeto con el mismo valor de “this”. Por ejemplo, si añadimos un objeto al mapa con this como hash, a no ser que conserváramos una referencia al objeto sería imposible eliminarlo pasandolo como argumento a remove.

El propio valor que se retorne no es muy importante porque habrá de ser procesado para evitar indexar con un número negativo o mayor que la propia tabla, pero si ha de ser rápidamente calculado.

Es importantísimo que los valores devueltos sean distribuidos uniformemente entre toda la tabla. Si llegarán a acumularse en una región determinada la búsqueda secuencial realizada en una misma posición daría al traste con la rapidez que se supone a tabla hash.

26.Recursos

* Thinking in Java 2ª ed. Uno de los mejores libros que el dinero puede comprar. Antes de hacerlo puedes visitar la sede de Bruce Eckel en artima.com o leerlo en linea desde codeguru.earthweb.com
* El Tutorial de Java en Sun esta mejorando con cada edición.
* Richard G. Balwin ha escrito sobre Java mas de lo que yo pueda leer en earthweb
* La API es necesaria para programar en Java.
* IBM también aporta un tutorial sobre Colecciones.