Para ver una copia completa de la licencia, acudir a la dirección
http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es

(9)
colecciones de datos

esquema de la unidad

(9.1) introducción a las colecciones de datos

En los lenguajes clásicos como **Pascal** o **C** por ejemplo siempre se ha distinguido entre estructuras de datos **estáticas** y **dinámicas**.

De modo que las estructuras de datos dinámicas se guardan en la zona de memoria convencional y tienen la particularidad de que se trata de datos cuyo tamaño se conoce a priori, es decir en tiempo de compilación. El ejemplo típico es el de los arrays clásicos (no los que se usan en Java), en ellos el tamaño de un array no puede modificarse una vez definido, por lo que no son la estructura apropiada para almacenar información que va creciendo o disminuyendo en tiempo de ejecución.

Las estructuras dinámicas se almacenan en la zona conocida como montículo de la memoria (**heap**) y su tamaño se puede reducir o aumentar en tiempo de ejecución.

Lo cierto es que en Java prácticamente todo es dinámico; incluso a los arrays se puede modificar su tamaño durante la ejecución del programa. De hecho todo lo que se puede crear con **new**, sería dinámico.

En el lenguaje C, las estructuras dinámicas se manejan mediante punteros que señalan a una zona de memoria previamente reservada con la función **malloc**, operador que permite reservar memoria dinámica bajo demanda, es decir se crea y elimina durante la ejecución del programa. Eso ocurre en casi todos los lenguajes clásicos. Las estructuras dinámicas han sido siempre utilizadas para crear colecciones de información que puede aumentar o disminuir durante la ejecución. A esas colecciones se las conoce como:

- **Listas** cuando es una enumeración de registros de datos los cuales pueden ser eliminados o aumentados en cualquier momento. Las hay **simplemente enlazadas** que sólo se pueden recorrer desde el principio al final y sólo en esa dirección; las **doblemente enlazadas** permiten recorrer la lista en cualquier dirección. Hay también **listas circulares** en las que no hay un nodo final de la lista porque éste apunta al primero de la lista generando una lista sin fin.
- **Pilas** cuando los datos se almacenan de modo que la última información añadida a la pila es la primera que se obtendrá cuando se recupere información de la pila (se las conoce también como estructuras **LIFO**, ***Last In First Out***, el último que entra es el primero que sale).
- **Colas** se trata de una estructura en las que los datos se almacenan de modo que al obtener información se consigue obtener el primer dato de la cola (estructura **FIFO**, ***First In First Out***, el primer que entra es el primero que sales).
- **Árboles**, en este caso los datos se enlazan formando una estructura de en forma de árbol invertido (al estilo de la estructura de los directorios y archivos). Se utiliza mucho para conseguir que los datos estén permanentemente ordenados

- **Tablas Hash**, se utilizan para conseguir índices que permiten asociar claves con valores. Son muy utilizadas en las bases de datos
- **Grafos**. Se llama así a cualquier estructura de datos donde la información se relaciona de forma libre (sin seguir ninguna de las estructuras anteriores).

En Java sin embargo existen una serie de clases e interfaces ya creadas que permiten manipular estas estructuras de forma más cómoda. Esas estructuras que representan fundamentalmente listas permiten implementar asociaciones entre clases cuya cardinalidad no es un número conocido sino que es indefinido, ejemplo:

Ilustración 9-1, La cardinalidad *asterisco* se implemente siempre mediante colecciones de datos. En este caso la clase Polígono crear internamente (al ser una composición) una lista de Vértices (de los que no puede haber menos de tres)

(9.2) jerarquía de clases Java para implementar colecciones

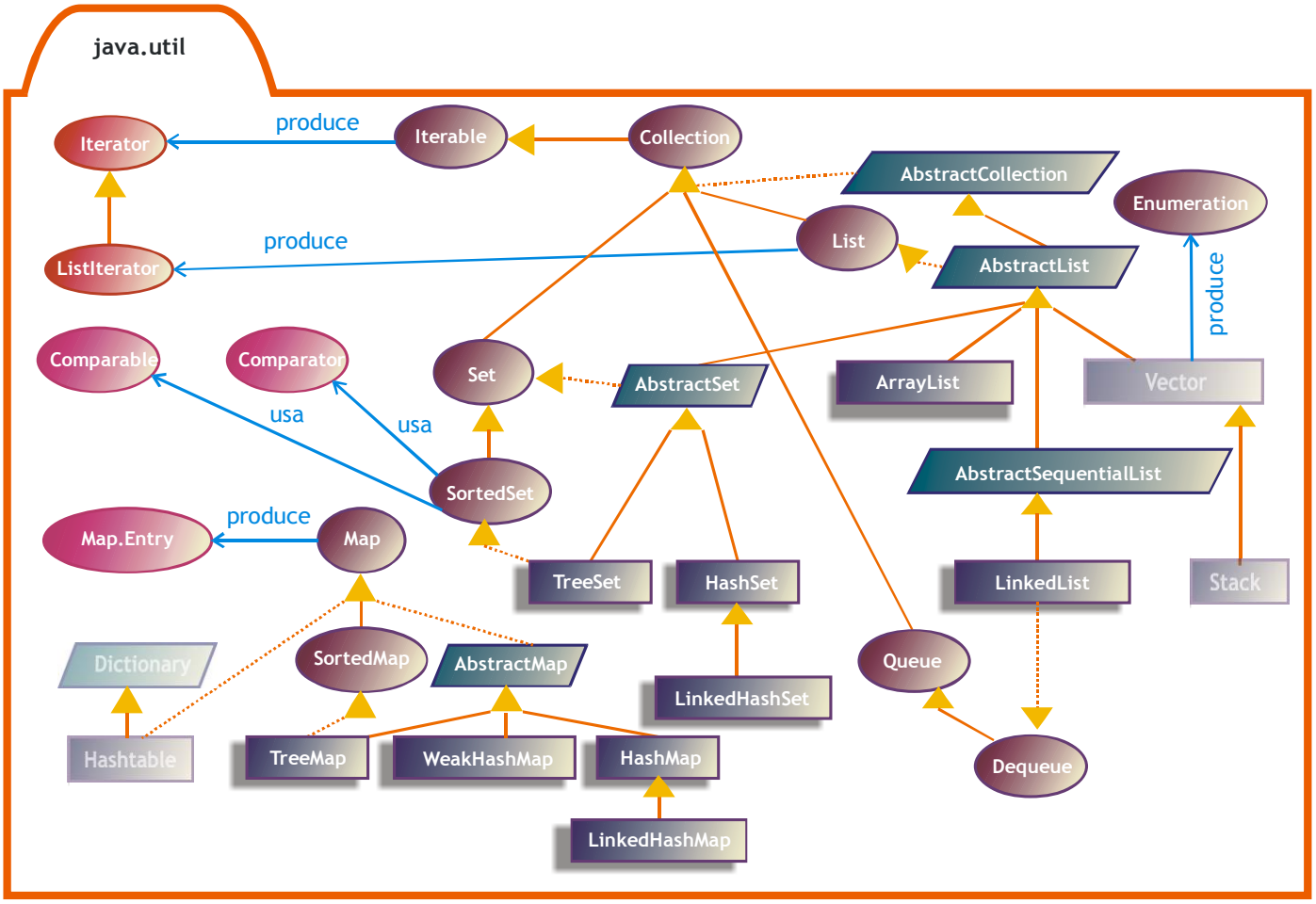

Ilustración 9-2, Diagrama de clases UML conteniendo los principales elementos de manejo de colecciones

EL número de interfaces, clases abstractas y clases disponibles para la creación de colecciones es espectacular. Poco a poco se detallarán las principales, aunque es interesante tener en mente el esquema anterior.

Por otro lado hay que tener en cuenta que en la versión 1.5 ha cambiado mucho el funcionamiento de estas interfaces y clases, además de añadir nuevas (como **Queue** o **Dequeue** por ejemplo).

Inicialmente se comentará el funcionamiento de las colecciones tal cual era en la versión 1.2, más adelante se comentarán las mejoras, en especial el funcionamiento de los tipos genéricos.

(9.3) interfaz Collection, listas simplemente enlazadas

La interfaz fundamental de trabajo con estructuras dinámicas es java.util.**Collection**. Es la raíz del funcionamiento de las colecciones y representa objetos que tienen la capacidad de almacenar listas de otros objetos. Esta interfaz define métodos muy interesantes para trabajar con listas que diversas clases implementan. Entre ellos:

método	uso
boolean **add**(Object o)	Añade el objeto a la colección. Devuelve **true** si se pudo completar la operación. Si no cambió la colección como resultado de la operación devuelve **false**
boolean **remove**(Object o)	Elimina al objeto indicado de la colección.
int **size**()	Devuelve el número de objetos almacenados en la colección
boolean **isEmpty**()	Indica si la colección está vacía
boolean **contains**(Object o)	Devuelve **true** si la colección contiene al objeto indicado *o*
void **clear**()	Elimina todos los elementos de la colección
boolean **addAll**(Collection otra)	Añade todos los elementos de la colección *otra* a la colección actual
boolean **removeAll**(Collection otra)	Elimina todos los objetos de la colección actual que estén en la colección *otra*
boolean **retainAll**(Collection otra)	Elimina todos los elementos de la colección que no estén en la otra
boolean **containsAll**(Collection otra)	Indica si la colección contiene todos los elementos de otra
Object[] **toArray**()	Convierte la colección en un array de objetos.
Iterator **iterator**()	Obtiene el objeto iterador de la colección, se explica en el punto siguiente

Hay que tener en cuenta que en la interfaz se considera que las colecciones las forman objétos genéricos (de la clase **Object**) que al ser padre de cualquier clase permite manipular sin problemas objetos del tipo que sea; no obstante habrá que hacer continuos *castings* para convertir tipos genéricos en el tipo concreto a manipular (salvo que utilicemos los genéricos de la versión 1.5 de Java y que se explican más adelante).

(9.3.1) iteradores

La interfaz **Iterator** (también en **java.util**) define objetos que permiten recorrer los elementos de una colección. Los métodos definidos por esta interfaz son:

método	uso
Object **next()**	Obtiene el siguiente objeto de la colección. Si se ha llegado al final de la colección y se intenta seguir, da lugar a una excepción de tipo: **NoSuchElementException** (que deriva a su vez de **RunTimeException**)
boolean **hasNext()**	Indica si hay un elemento siguiente (y así evita la excepción).
void **remove()**	Elimina el último elemento devuelto por **next**

Ejemplo (recorrido por una colección):

```
//suponiendo que colecciónString es una colección de textos
Iterator it=colecciónString.iterator();
while(it.hasNext()){
   String s=(String)it.next();//puesto que next devuelve objetos
                              //Object, es necesario el casting
   System.out.println(s);
}
```

La interfaz **Collection** junto con los iteradores proporcionados por la interfaz **Iterator** permiten crear listas simplemente enlazadas. Es decir listas cuyos nuevos miembros se añaden al final de la lista (mediante el método **add** de la clase **Collection**) y que sólo se pueden recorrer de izquierda a derecha.

Estas listas en todo momento mantienen el orden en el que fueron insertados los datos; aunque sí permiten eliminar datos del interior de la lista (método **remove** tanto de la interfaz **Collection** como de la interfaz **Interator**).

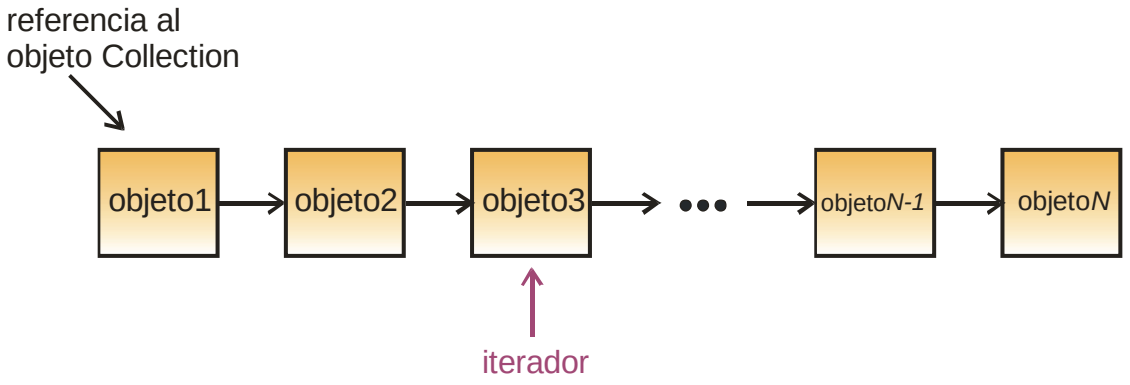

Ilustración 9-3, Representación gráfica de una lista simplemente enlazada (que es lo que implementa la clase Collection), el iterador en todo momento señala a un objeto de la lista, pero sólo puede avanzar de izquierda a derecha en la lista

Sin embargo las colecciones genéricas de Java no admiten añadir elementos entre dos existentes, para eso se utilizan las colecciones de tipo **List** (es decir se parecen a las colas).

(9.3.2) for each para colecciones

Existe una versión de instrucción **for**, disponible desde la versión 1.5 de Java que permite usar de forma más fácil un bucle de recorrido por una colección de la misma forma que hace un objeto iterador. En concreto la sintaxis es:

```
for(Object objto:nombreColección){
    ...
}
```

Este bucle **for** recorre cada elemento de la colección y en cada vuelta el siguiente elemento de la colección se asigna al objeto indicado. Por ejemplo el código iterador indicado anteriormente, ahora se podría hacer así:

```
//suponiendo que coleccionString es una colección de textos
for(Object o:coleccionString){
    String s=(String)o;
    System.out.println(s);
}
```

Hay que tener en cuenta que sólo es posible utilizar este bucle si deseamos avanzar por todos los elementos de una lista. Pero puesto que se trata de algo muy habitual, es un bucle muy utilizado.

(9.4) interfaz List, listas doblemente enlazadas

(9.4.1) interfaz List

List es una interfaz (de java.util) que se utiliza para definir listas doblemente enlazadas. En este tipo de listas importa la posición de los objetos, de modo que se pueden recolocar. Deriva de la interfaz **Collection** por lo que tiene disponible todos sus métodos, pero además aporta métodos mucho más potentes:

método	uso
void **add**(int índice, Object elemento)	Añade el elemento indicado en la posición *índice* de la lista
void **remove**(int índice)	Elimina el elemento cuya posición en la colección la da el parámetro *índice*

método	uso
Object **set**(int índice, Object elemento)	Sustituye el elemento número *índice* por uno nuevo. Devuelve además el elemento antiguo
Object **get**(int índice)	Obtiene el elemento almacenado en la colección en la posición que indica el índice
int **indexOf**(Object elemento)	Devuelve la posición del elemento. Si no lo encuentra, devuelve -1
int **lastIndexOf**(Object elemento)	Devuelve la posición del elemento comenzando a buscarle por el final. Si no lo encuentra, devuelve -1
void **addAll**(int índice, Collection elemento)	Añade todos los elementos de una colección a una posición dada.
ListIterator **listIterator**()	Obtiene el iterador de lista que permite recorrer los elementos de la lista
ListIterator **listIterator**(int índice)	Obtiene el iterador de lista que permite recorrer los elementos de la lista. El iterador se coloca inicialmente apuntando al elemento cuyo índice en la colección es el indicado.
List **subList**(int desde, int hasta)	Obtiene una lista con los elementos que van de la posición *desde* a la posición *hasta*

Cualquier error en los índices produce **IndexOutOfBoundsException**

(9.4.2) ListIterator

Es un interfaz que define clases de objetos para recorrer listas. Es heredera de la interfaz **Iterator**. Aporta los siguientes métodos

método	uso
void **add**(Object elemento)	Añade el elemento delante de la posición actual del iterador
void **set**(Object elemento)	Sustituye el elemento señalado por el iterador, por el elemento indicado
Object **previous**()	Obtiene el elemento previo al actual. Si no lo hay provoca excepción: **NoSuchElementException**
boolean **hasPrevious**()	Indica si hay elemento anterior al actualmente señalado por el iterador
int **nextIndex**()	Obtiene el índice del elemento siguiente
int **previousIndex**()	Obtiene el índice del elemento anterior

Los iteradores de este tipo son mucho más potentes que los de tipo **Iterator** ya que admiten recorrer la lista en cualquier dirección e incluso ser utilizados para modificar la lista. Además copntiene todos los métodos de **Iterator** (como **remove** por ejemplo) ya que los hereda.

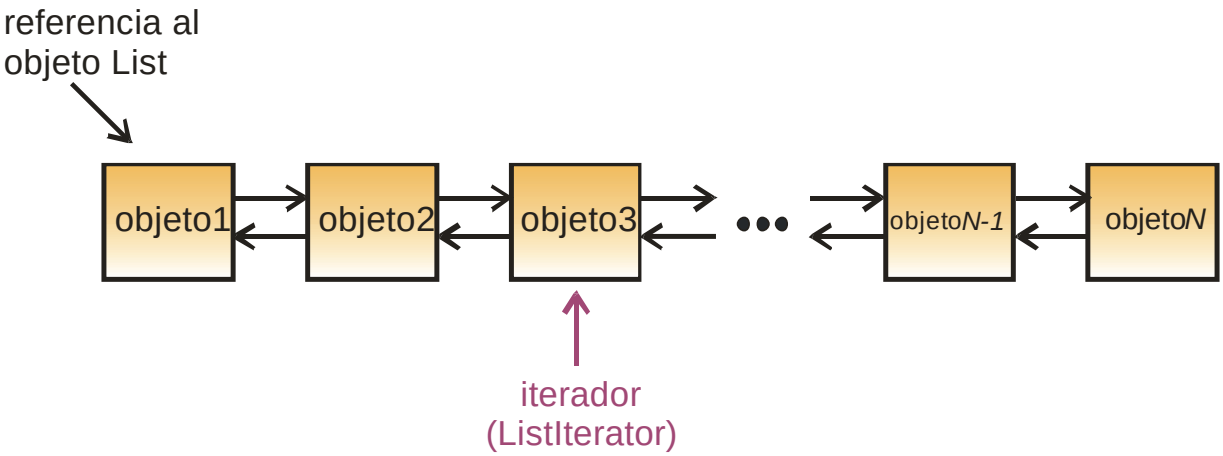

Ilustración 9-4, Representación gráfica de una colección de datos de tipo List. El iterador puede recorrer la lista en cualquier dirección

(9.4.3) clase ArrayList

Implementa la interfaz **List**. Está pensada para crear tanto listas simplemes como doblemente enlazadas. Está disponible desde la versión 1.2 y es la clase fundamental para representar colecciones de datos

Posee tres constructores:

- **ArrayList().** Constructor por defecto. Simplemente crea un ArrayList vacío
- **ArrayList(int** capacidadInicial**).** Crea una lista con una capacidad inicial indicada.
- **ArrayList(Collection** c**).** Crea una lista a partir de los elementos de la colección indicada.

Ejemplo:

```
ArrayList a=new ArrayList();
a.add("Hola");
a.add("Adiós");
a.add("Hasta luego");
a.add(0,"Buenos días");
for(Object o:a){
    System.out.println(o);
}
/*escribe:
            Buenos días
            Hola
            Adiós
            Hasta luego
*/
```

(9.4.4) clase LinkedList

Es una clase heredera de las anteriores e implementa métodos que permiten crear listas de adición tanto por delante como por detrás (**listas dobles**). Desde este clase es sencillo implantar estructuras en forma de pila o de cola. Añade los métodos:

método	uso
Object **getFirst()**	Obtiene el primer elemento de la lista
Object **getLast()**	Obtiene el último elemento de la lista
void **addFirst(**Object o**)**	Añade el objeto al principio de la lista
void **addLast(**Object o**)**	Añade el objeto al final de la lista
void **removeFirst()**	Borra el primer elemento
void **removeLast()**	Borra el último elemento

Los métodos están pensados para que las listas creadas mediante objetos **LinkedList** sirven para añadir elementos por la cabezo o la cola, pero en ningún caso por el interior.

En el caso de implementar pilas, los nuevos elementos de la pila se añadirían por la cola (mediante **addLast**) y se obtendrían por la propia cola (**getLast**, **removeLast**).

En las colas, los nuevos elementos se añaden por la cola, pero se obtienen por la cabeza (**getFirst**, **removeFirst**).

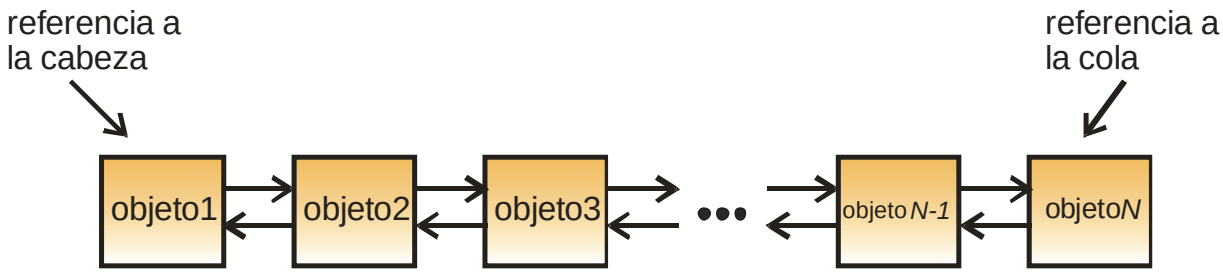

Ilustración 9-5, Representación gráfica de una colección de datos de tipo LinkedList. La idea es no utilizar iteradores y manejar sólo los elementos de la cabeza y la cola-

(9.5) genéricos de Java

(9.5.1) introducción a los genéricos

Se trata de la implementación en Java de una de las opciones de C++ más añoradas por los programadores que proceden de ese lenguaje; las **plantillas** (***templates***). Son similares a las plantillas de C++. Sirven para evitar conversiones de clases.

Probablemente sin la existencia de las colecciones no habrían sido implementados. Pero las colecciones al ser formadas de objetos de tipo **Object**, tienen una serie de problemas que hasta la versión 1.5 fueron sufridas por los programadores/as:

- Nada impide crear listas heterogéneas de objetos (por ejemplo colecciones de objetos String junto con Arrays de enteros). Eso causa problemas evidentes de casting, además de posibles incoherencias.
- El que las listas sean siempre de tipo Object, provocan una cantidad grande de conversiones en el código (mediante casting), haciéndole más pesado de comprender.
- Los métodos de las clases e interfaces de las colecciones no se adaptan al tipo de datos que contienen dificultando su uso.

Por ello aparecieron los tipos genéricos en la versión 1.5 de Java consiguiendo solucionar los problemas anteriores en las colecciones.

(9.5.2) idea general de los genéricos

Los genéricos permiten crear clases y métodos en los que los tipos de datos (las clases) sobre los que actúan son un parámetro más. Gracias a ello se pueden crear clases y métodos que se adaptan a distintos tipos de datos de forma automática.

Anteriormente a la aparición de los genéricos el que las clases y métodos se adaptaran automáticamente pasaba por especificar parámetros de tipo **Object**, ya que las referencias de tipo Object valen para cualquier otro tipo de objeto.

El problema es que la coherencia no está asegurada con los tipos Object, la sintaxis del lenguaje puede provocar excepciones de tipo **ClassCastException** ante conversiones erróneas de Object a otro tipo de datos.

La idea es:

(1) El programar declara un genérico en un método o una clase (es decir, le da nombre a un tipo de clases que representa a cualquier tipo), por ejemplo **T**

(2) Cuando se utilice dicha clase o método se indicará un tipo concreto (por ejemplo **String**), entonces se cambia el genérico (todas las **T**) para que ahora se refiera a **String**.

(9.5.3) genéricos en métodos

Se pueden indicar genéricos en los métodos. Para indicar que el método se puede utilizar con cualquier tipo de objeto y que el método se adaptará a dicho tipo. La sintaxis es:

modificadoresDel Método <**Genérico**> tipoDevuelto nombreMétodo(...)

Ejemplo para un método capaz de obtener de forma aleatoria un elemento de un array del tipo que sea:

```
public class UtilesArrays {
    public static <T> T tomarAleatorio(T a[]){
            int i=(int)(Math.random()*a.length);
            return a[i];
    }
}
```

En el ejemplo la letra *T* entre los signos < y > indica que se va a utilizar un tipo genérico, el hecho de que le siga otra *T* significa que el tipo que devuelve el método *tomarAleatorio* será el mismo que el del parámetro *a*, y se corresponderá al tipo genérico *T*. Después cuando se invoque a este método, la letra genérica *T* será traducida por el tipo con el que se invoque al método. Por ejemplo desde este código:

```
public static void main(String[] args) {
    String a[]={"a","b","c","d","e","f"};
    String toma=UtilesArrays.tomarAleatorio(a);
    System.out.println(toma);
}
```

En ese código, como se invoca usando un array de **Strings**, por lo tanto la letra genérica *T* será traducida por *String*. Es decir es como si el código anterior se tradujera como:

```
public class UtilesArrays {
    public static String tomarAleatorio(String a[]){
            int i=(int)(Math.random()*a.length);
            return a[i];
    }
}
```

Lo interesante es que el código que invoca el método con genéricos es más cómodo y fácil de entender.

(9.5.4) genéricos en las clases

Sin embargo el método habitual de utilizar genéricos, debido a su mayor potencia, es declarar el genérico en una clase. De esta forma indicados que el código de la clase utiliza tipos genéricos que serán traducidos por el tipo de datos que se desee.

Por ejemplo una clase pensada para extraer elementos aleatorios de un array del tipo que sea se podría declarar así:

```
public class EltoAleatorio <T>  {
    T elemento;

    public EltoAleatorio(T array[]) {
            elemento=array[(int)(Math.random()*array.length)];
    }

    public T getElemento() {
            return elemento;
    }
}
```

Los tipos se indican tras el nombre de la clase. De esta forma cuando se creen objetos de clase *EltoAleatorio* habrá que indicar la clase por la que se traducirá el genérico *T*. Por ejemplo:

```
public static void main(String[] args) {
    String a[]={"uno","dos","tres","cuatro"};
    EltoAleatorio<String> e=new EltoAleatorio<String>(a);

    System.out.println(e.getElemento());
}
```

Obsérvese que tanto al declarar el objeto *e* hay que concretar el valor del genérico de ahí que la declaración sea `EltoAleatorio<String> e` de ese modo ya queda claro la traducción de *T*. A partir de ese momento ya todos los métodos de clase y propiedades que utilizaran el genérico *T*, ahora utilizarán el tipo *String*.

En definitiva los genéricos de clase marcan las posiciones para la clase concreta que habrá que especificar en la creación del objeto.

Lo bueno es que pueden convivir distintas concreciones de genéricos, podría ser:

```
EltoAleatorio<String> e1=new EltoAleatorio<String>(a1);
EltoAleatorio<Integer> e2=new EltoAleatorio<Integer>(a2);
```

En ambos casos el tipo es *EltoAleatorio*, pero son objetos que incluso se puede entender que tiene tipos distintos ya que en el primero el genérico se concretará en **String** y en el segundo por **Integer**. Es más, podemos decir que

el tipo del primero es *EltoAleatorio<String>* y el segundo de tipo *EltoAleatorio<Integer>*.

(9.5.5) interfaces con genéricos

Al igual que las clases, las interfaces pueden utilizar genéricos en las mismas condiciones que en éstas:

```
public interface Compar  <T>{
    T menor();
    T mayor();
}
```

Una clase que implemente la interfaz puede hacerlo así:

```
public class Clase1 implements Compar<String>{
```

De modo que traduce el genérico de la interfaz por un tipo concreto o bien:

```
public class Clase1 <T> implements Compar<T>{
```

De modo que no traduce el genérico y espera a que durante la creación de objetos de esa clase se indique el tipo concreto para el genérico.

(9.5.6) uso de varios genéricos

tanto en métodos, como en clases o interfaces, es posible utilizar dos tipos genéricos e incluso más. Para ello se separan por comas dentro de los signos < y >. Ejemplo:

```
public class ComparadorObjs <T1,T2>{
    protected T1 obj1;
    protected T2 obj2;
    public ComparadorObjs(T1 obj1,T2 obj2) {
            this.obj1=obj1;
            this.obj2=obj2;
    }

    public boolean mismaPrimeraLetra(){
            String letra1=obj1.getClass().getName().substring(1,1);
            String letra2=obj2.getClass().getName().substring(1,1);
            return letra1.equalsIgnoreCase(letra2);
    }
}
```

Esta clase (que en realidad no es muy útil), construye un objeto a partir de otros dos. El primero de tipo genérico *T1*, y el segundo de tipo *T2*. El método *mismaPrimeraLetra* devuelve verdadero si el nombre de clase de *T1* empieza por la misma letra que *T2*.

Para utilizar esta clase, ejemplo:

```
public static void main(String[] args) {
    String s1="Hola";
    StringBuffer s2=new StringBuffer("Adiós");
    ComparadorObjs<String, StringBuffer> comp=
            new ComparadorObjs<String, StringBuffer>(s1,s2);

    System.out.println(comp.mismaPrimeraLetra());
}
```

Devolverá verdadero porque tanto **String** como **StringBuffer** empiezan por *S*. En el ejemplo *T1* se convertirá en *String* y *T2* en *StringBuffer*.

(9.5.7) limitación de los tipos posibles en un genérico

A veces no interesa que las clases acepten a cualquier tipo de objeto, sino a objetos de un determinado tipo y sus descendientes. Eso es posible indicarlo mediante la palabra **extends** dentro de la declaración del genérico. Ejemplo:

```
public class ManejadorVehiculos <V extends Vehiculo>{
    protected V vehiculo;
    public ManejadorVehiculos(V vehi) {
            vehiculo=vehi;
    }
    ....
            vehiculo.arrancar();
    ....
}
```

En el ejemplo a la clase se le puede indicar cualquier clase descendiente de *Vehiculo*, pero ninguna otra. Por ejemplo:

```
ManejadorVehiculos v1<Coche>=new
                            ManejadorVehiculos<Coche>(c);
ManejadorVehiculos v2<Autocar>=new
                            ManejadorVehiculos<Autocar>(d);
```

Sin embargo **no** se podría declarar:

```
ManejadorVehiculos v3<String>=new
                            ManejadorVehiculos<String>(e);
```

No es posible porque la clase String no es heredera de ningún vehículo.

La razón de utilizar esta cláusula tan restrictiva está en el hecho de poder realizar acciones que sabemos que sólo son posibles en cierto tipo de objetos. La clase *ManejadorVehiculos* necesita arrancar el vehículo, por ello tenemos que asegurar que el genérico no puede ser ninguna clase incompatible.

Por otro lado **extends** no tiene por qué referirse a clases, también puede referirse a interfaces. Ejemplo:

```
public class Clase1 <V extends Comparable>{
```

Significa que la *Clase1* usa un genérico al que se le puede asignar cualquier clase que implemente la interfaz **Comparable**.

(9.5.8) tipos comodín

Aunque parece que todo está resuelto de esta forma. Hay problemas cuando mezclamos objetos de la misma clase pero distinta traducción de genérico. Es el caso de este ejemplo:

```
public class EltoAleatorio <T>  {
    T elemento;
    int índice;

    public EltoAleatorio(T array[]) {
            índice=(int)(Math.random()*array.length);
            elemento=array[índice];
    }

    public T getElemento() {
            return elemento;
    }

    public boolean mismoÍndice(EltoAleatorio<T> elto){
            return elto.índice==this.índice;
    }
}
```

En este caso a la clase *EltoAleatorio* se le ha añadido una propiedad que almacena el índice aleatorio que obtiene el constructor. Eso permite que construyamos un método llamado *mismoÍndice* que recibe un objeto de clase *EltoAleatorio* y nos dice si el índice aleatorio calculado fue el mismo. Para usar este método:

```
String s1 []={"a","b","c","d","e","f"};
String s2 []={"1","2","3","4","5","6"};
EltoAleatorio<String> e1=new EltoAleatorio<String>(s1);
EltoAleatorio<String> e2=new EltoAleatorio<String>(s2);
System.out.println(e1.mismoÍndice(e2));
```

El código funciona, sólo devolverá verdadero si tanto en el objeto *e1* como en *e2*, el índice tiene el mismo valor.

Sin embargo, este otro código falla:

```
String s1 []={"a","b","c","d","e","f"};
Integer s2 []={1,2,3,4,5,6};
EltoAleatorio<String> e1=new EltoAleatorio<String>(s1);
EltoAleatorio<Integer> e2=new EltoAleatorio<Integer>(s2);
System.out.println(e1.mismoÍndice(e2));
```

El error ocurre en tiempo de compilación. La razón, que en la línea remarcada el objeto *e1* es ya de tipo *EltoAleatorio<String>* por lo que el método *mismoÍndice* sólo puede aceptar objetos de tipo *EltoAleatorio<String>*, ya que el genérico T se tradujo como String. El problema pues estriba en que dicho método tiene que poder aceptar cualquier tipo de objeto de tipo *EltoAleatorio*.

Para ello se usa el signo **<?>** de ese modo indicamos la posibilidad de aceptar cualquier tipo de clase con genérico. Es decir el método se reescribiría así:

```
public boolean mismoÍndice(EltoAleatorio<?> elto){
    return elto.índice==this.índice;
}
```

De esa forma se indica que *elto* es un objeto de tipo *EltoAleatorio* tenga el tipo que tenga el genérico. El ejemplo anterior que fallaba ahora funcionará.

Incluso se puede delimitar el genérico:

```
public boolean mismoÍndice(EltoAleatorio<? extends Number> elto){
    return elto.índice==this.índice;
}
```

Ahora el método acepta cualquier tipo de *EltoAleatorio* pero siempre y cuando se el tipo genérico halla sido traducido por cualquier clase heredera de **Number**.

(9.5.9) genéricos en las colecciones

Las colecciones aglutinan objetos de un mismo tipo desde la aparición de Java 1.5; en esta versión se abandona el hecho de considerar que los elementos de una colección referencias de tipo **Object** y se redefinieron todas las interfaces y clases para que utilizaran genéricos.

El código anterior sigue siendo compatible ya que cuando no se especifica la clase a adoptar por el genérico, se toma Object. Pero lo recomendable es adaptarse a la nueva sintaxis.

Así la interfaz **Collection** ahora se llama **Collection<E>** porque utiliza genéricos y sus métodos también les utiliza.

Lo que varía en definitiva es que al indicar ahora colecciones e iteradores del tipo que sean, hay que indicar también la traducción de sus genéricos. Dicho de otra forma, hay que indicar el tipo de lista o de iterador que estamos creando. Por ejemplo:

```
ArrayList<String> lista=new ArrayList<String>();
ListIterator<String> iterador=lista.listIterator();
```

A continuación se indica los métodos de las interfaces y clases vistas anteriormente, pero tal cual quedan en la versión 1.5

métodos de Collection<E>

Hay que tener en cuenta, según lo comentado anteriormente que *E* es el tipo genérico de la colección.

método	uso
boolean **add**(E o)	Añade el objeto a la colección. Devuelve **true** si se pudo completar la operación. Si no cambió la colección como resultado de la operación devuelve **false**
boolean **remove**(E o)	Elimina al objeto indicado de la colección.
int **size**()	Devuelve el número de objetos almacenados en la colección
boolean **isEmpty**()	Indica si la colección está vacía
boolean **contains**(Object o)	Devuelve **true** si la colección contiene al objeto indicado
void **clear**()	Elimina todos los elementos de la colección
boolean **addAll**(Collection<? extends E otra)	Añade todos los elementos de la colección *otra* a la colección actual. Sólo puede añadir objetos de colecciones cuyos elementos sean de tipos compatibles con el actual.
boolean **removeAll**(Collection<?extends E> otra)	Elimina todos los objetos de la colección actual que estén en la colección *otra*
boolean **retainAll**(Collection<? extends E otra)	Elimina todos los elementos de la colección que no estén en la otra
boolean **containsAll**(Collection<? extends E otra)	Indica si la colección contiene todos los elementos de otra
Object[] **toArray**()	Convierte la colección en un array de objetos.
<T>T[] **toArray**(T array)	Convierte la colección en un array de objetos. El array devuelto contiene todos los elementos de la colección, el array de devolución es del mismo tipo que el array que recibe de argumento (de hecho es la única utilidad que tiene este argumento, la de decir el tipo de array que se ha de devolver).

método	uso
Iterator<E> **iterator()**	Obtiene el objeto iterador de la colección, se explica en el punto siguiente

métodos de List<E>

método	uso
void **add**(int índice, E elemento)	Añade el elemento indicado en la posición *índice* de la lista
void **remove**(int índice)	Elimina el elemento cuya posición en la colección la da el parámetro *índice*
E **set**(int índice, E elemento)	Sustituye el elemento número *índice* por uno nuevo. Devuelve además el elemento antiguo
E **get**(int índice)	Obtiene el elemento almacenado en la colección en la posición que indica el índice
int **indexOf**(Object elemento)	Devuelve la posición del elemento. Si no lo encuentra, devuelve -1
int **lastIndexOf**(Object elemento)	Devuelve la posición del elemento comenzando a buscarle por el final. Si no lo encuentra, devuelve -1
void **addAll**(int índice, Collection <? extends E> elemento)	Añade todos los elementos de una colección compatible a partir de la posición dada.
List<E> **subList**(int inicio, int fin)	Devuelve una lista del mismo tipo que la actual tomando los elementos que van desde el inicio indicado al fin (sin incluir este último).
ListIterator<E> **listIterator()**	Obtiene el iterador de lista que permite recorrer los elementos de la lista
ListIterator<E> **listIterator**(int índice)	Obtiene el iterador de lista que permite recorrer los elementos de la lista. El iterador se coloca inicialmente apuntando al elemento cuyo índice en la colección es el indicado.

métodos de Iterator<E>

método	uso
E **next()**	Obtiene el siguiente objeto de la colección. Si se ha llegado al final de la colección y se intenta seguir, da lugar a una excepción de tipo: **NoSuchElementException** (que deriva a su vez de **RunTimeException**)
boolean **hasNext()**	Indica si hay un elemento siguiente (y así evita la excepción).
void **remove()**	Elimina el último elemento devuelto por **next**

métodos de ListIterator<E>

método	uso
void add(E elemento)	Añade el elemento delante de la posición actual del iterador
void set(E elemento)	Sustituye el elemento señalado por el iterador, por el elemento indicado
E previous()	Obtiene el elemento previo al actual. Si no lo hay provoca excepción: NoSuchElementException
boolean hasPrevious()	Indica si hay elemento anterior al actualmente señalado por el iterador
int nextIndex()	Obtiene el índice del elemento siguiente
int previousIndex()	Obtiene el índice del elemento anterior

métodos de LinkedList<E>

método	uso
E getFirst()	Obtiene el primer elemento de la lista
E getLast()	Obtiene el último elemento de la lista
void addFirst(E o)	Añade el objeto al principio de la lista
void addLast(E o)	Añade el objeto al final de la lista
void removeFirst()	Borra el primer elemento
void removeLast()	Borra el último elemento

ejemplo de uso de recorrido de lista usando genéricos

```
ArrayList<String> lista=new ArrayList<String>();
lista.add("Hola");
lista.add("Adiós");
lista.add("Hasta luego");
lista.add("Ciao");
Iterator<String> it=lista.iterator();
while(it.hasNext()){
    String s=it.next();
    System.out.println(s);
}
```

Se observa la ausencia de castings en el código. Usando el bucle **for..each** el código sería:

```
ArrayList<String> lista=new ArrayList<String>();
lista.add("Hola");
lista.add("Adiós");
lista.add("Hasta luego");
lista.add("Ciao");
for (String s : lista) {
    System.out.println(s);
}
```

(9.6) colecciones sin duplicados

interfaz Set

Permite implementar listas dinámicas de elementos sin duplicados. Deriva de **Collection**. Es el método **equals** el que se encarga de determinar si dos objetos son duplicados en la lista (habrá que redefinir este método para que funcione adecuadamente).

Posee los mismos métodos que la interfaz **Collection**. La diferencia está en el uso de duplicados.

clase HashSet

Implementa la interfaz anterior. Por lo que es la clase más utilizada para implementar listas sin duplicados.

Tiene los métodos de la interfaz **List** (y por lo tanto de un objeto de tipo **ArrayList**) pero además utiliza internamente una tabla de tipo hash. Esas tablas asocian claves a conjuntos de valores

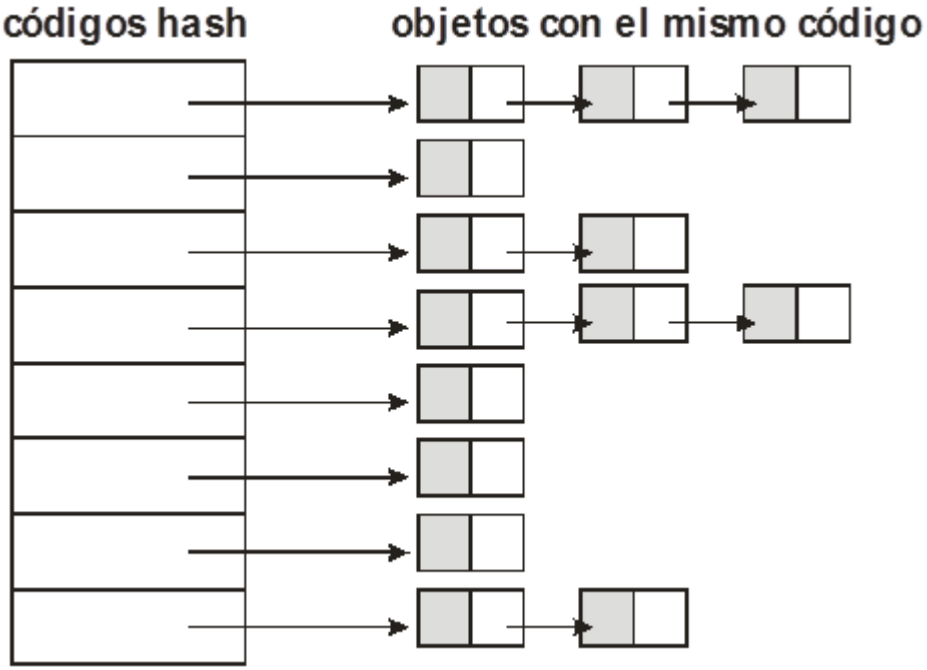

Ilustración 9-6, Ejemplo de tabla hash, varios elementos se asocian con el mismo código. La tabla (un array) mantiene los códigos unicos por un lado y en cada hay una lista de los objetos con dicho código

La naturaleza de las tablas hash hace que cuando se crean listas **HashSet**, no habrá valores duplicados, pero en absoluto se garantiza el orden. Es decir cada vez que llega un valor único al array se añade en una posición del mismo (la siguiente que esté libre), si llega otro con el mismo valor se añade a la lista de esa celda del array.

Los objetos HashSet se construyen con un tamaño inicial de tabla (el tamaño del array) y un factor de carga que indica cuándo se debe redimensionar el array. Es decir si se creó un array de 100 elementos y la carga se estableció al 80%, entonces cuando se hayan rellenado 80 valores únicos, se redimensiona el array.

Por defecto el tamaño del array se toma con 16 y el factor de carga con 0,75 (75%). No obstante se puede construir una lista **HashSet** indicando ambos parámetros. Los posibles constructores de la clase **HashSet <E>** son:

Constructor	uso
HashSet()	Construye una nueva lista vacía con tamaño inicial 16 y un factor de carga de 0,75
HashSet(Collection <? extends E> lista)	Crea una lista Set a partir de la colección compatible indicada
HashSet(int capacidad)	Crea una lista con el tamaño indicado y un factopr de 0,16
HashSet(int capacidad, double factor)	Crea una lista con la capacidad y el factor indicados.

La cuestión es ¿cómo compara Java los objetos de la lista ara saber si son iguales? Bueno, pues la cuestión es que utiliza el método **equals** heredado de la clase base **Object**. Por ello es necesario que las clases de los objetos que se almacenarán en la tabla definan ese método. No es el único requisito, de hecho el fundamental es que se defina también el método heredado **hashCode**.

La razón es que es el **hashCode** es el código pensado para este tipo de lista, de hecho es el identificador en una lista HashSet, por ello los objetos que consideremos iguales en contenido deben devolver el mismo hashCode, es decir el mismo número entero.

El método **equals** hay que definirlo para saber cuándo dos objetos de la lista son iguales. Un ejemplo de clase preparada para estas lides es:

```
public class Alumno {
    protected String nombre;
    protected double nota;
    //constructor
    public Alumno(String n, double nt){
            nombre=n;
            nota=nt;
    }
    public boolean equals(Object obj) {
            if(obj instanceof Alumno){
                    Alumno a=(Alumno) obj;
                    return a.nombre.equals(nombre) && a.nota==nota;
            }
            else return false;
    }
```

```
    public int hashCode() {
            return nombre.hashCode()+(int)nota*10000;
    }
    public String toString() {
            return nombre+"-"+nota;
    }
}
```

Un ejemplo de uso de lista de este tipo sería:

```
Alumno a1=new Alumno("alberto",7);
Alumno a2=new Alumno("alberto",6);
Alumno a3=new Alumno("alberto",7);
Alumno a4=new Alumno("adrian",7);
Alumno a5=new Alumno("alberto",7);
Alumno a6=new Alumno("adrian",8);
HashSet<Alumno> l=new HashSet<Alumno>();
l.add(a1);
l.add(a2);
l.add(a3);
l.add(a4);
l.add(a5);
l.add(a6);
for (Alumno alumno : l) {
    System.out.println(alumno);
}
/* sale:
            adrian-8.0
            adrian-7.0
            alberto-6.0
            alberto-7.0
*/
```

Los valores repetidos no se muestran. De hecho no funciona realmente como una tabla hash teórica, sino como una lista de valores únicos. En cuanto borremos un elemento, ese valor desaparece sin importar que hayamos insertado cinco objetos con el mismo valor.

(9.6.1) clase LinkedHashSet

Se trata de una clase heredera de la anterior con los mismos métodos y funciones, pero que consigue mantener en el orden en el que los datos fueron insertados, es decir utilizando la clase alumnos la salida de este código (a diferencia del obtenido con la lista de tipo HashSet sería):

```
Alumno a1=new Alumno("alberto",7);
Alumno a2=new Alumno("alberto",6);
Alumno a3=new Alumno("alberto",7);
Alumno a4=new Alumno("adrian",7);
Alumno a5=new Alumno("alberto",7);
Alumno a6=new Alumno("adrian",8);
LinkedHashSet<Alumno> l=new LinkedHashSet<Alumno>();
l.add(a1);
l.add(a2);
l.add(a3);
l.add(a4);
l.add(a5);
l.add(a6);
for (Alumno alumno : l) {
   System.out.println(alumno);
}
/* sale:
          alberto-7.0
          alberto-6.0
          adrian-7.0
          adrian-8.0
*/
```

(9.7) árboles. listas ordenadas

(9.7.1) árboles

Un árbol es una estructura en la que los datos se organizan en nodos los cuales se relacionan con dos o más nodos. En general se utilizan para ordenar datos y en ese caso de cada nodo sólo pueden colgar otros dos de modo que a la izquierda cuelgan valores menores y a la derecha valores mayores.

Al recorrer esta estructura, los datos aparecen automáticamente en el orden correcto. La adición de elementos es más lenta, pero su recorrido ordenado es mucho más eficiente.

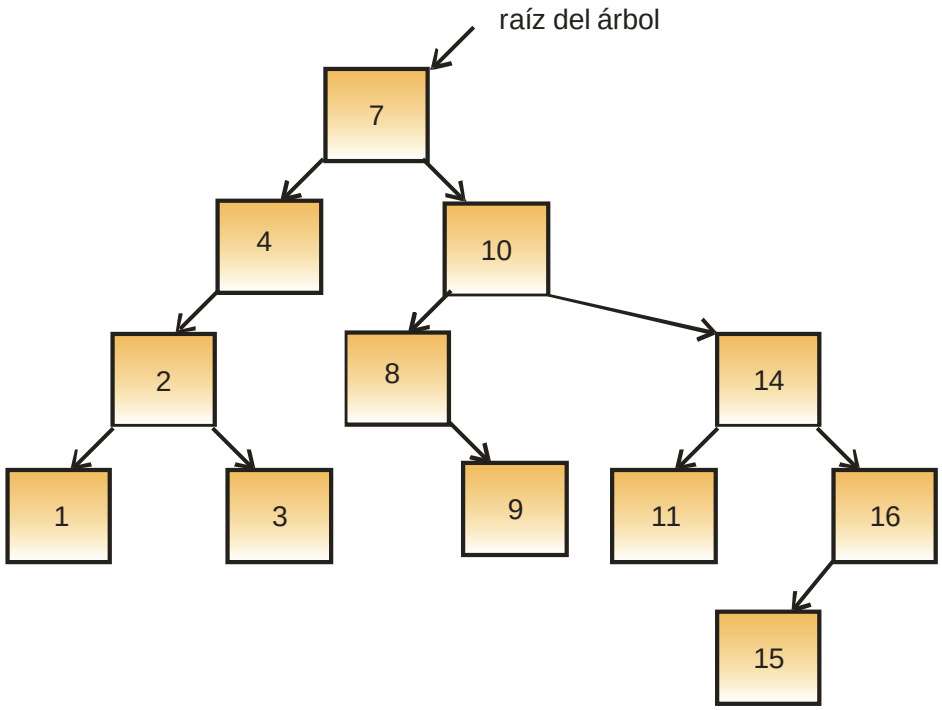

Ilustración 9-7, Representación gráfica del funcionamiento de un árbol binario ordenado.

(9.7.2) interfaz SortedSet<E>

La interfaz **SortedSet** es la encargada de definir esta estructura. Esta interfaz deriva de **Collection** y añade estos métodos:

método	uso
E **first()**	Obtiene el primer elemento del árbol (el más pequeño)
E **last()**	Obtiene el último elemento del árbol (el más grande)
SortedSet **headSet**(E o)	Obtiene un **SortedSet** que contendrá todos los elementos menores que el objeto ***o***.
SortedSet **tailSet**(E o)	Obtiene un **SortedSet** que contendrá todos los elementos mayores que el objeto ***o***.
SortedSet **subSet**(E menor, E mayor)	Obtiene un **SortedSet** que contendrá todos los elementos del árbol cuyos valores ordenados estén entre el menor y mayor objeto indicado
Comparator **comparator()**	Obtiene el objeto comparador de la lista

El resto de métodos son los de la interfaz **Collection** (sobre todo **add** y **remove**).

(9.7.3) clase TreeSet<E>

Se trata de la clase que se utiliza prioritariamente para conseguir árboles ordenados ya que implementa la interfaz anterior.

(9.7.4) comparaciones

El problema es que los objetos tienen que poder ser comparados para determinar su orden en el árbol. Esto implica implementar la interfaz **Comparable<E>** de Java (está en **java.lang**). Esta interfaz define el método **compareTo** que utiliza como argumento un objeto a comparar y que devuelve 0 si los objetos son iguales, un número positivo si el primero es mayor que el segundo y negativo en caso contrario.

Con lo cual los objetos a incluir en un TreeSet deben implementar **Comparable** y esto les obliga a redefinir el método compareTo (recordando que su argumento es de tipo **Object**).Ejemplo (usando la clase **Alumnos**):

```
public class Alumno implements Comparable<Alumno>{
    protected String nombre;
    protected double nota;
    ...
    public int compareTo(Alumno o) {
            int comparacion=nombre.compareToIgnoreCase(o.nombre);
            if(comparacion==0){
                    return (int)(nota-o.nota);
            }
            else return comparacion;
    }
    ...
}
.....................
public static void main(String[] args) {
    Alumno a1=new Alumno("alberto",7);
    Alumno a2=new Alumno("alberto",6);
    Alumno a3=new Alumno("alberto",7);
    Alumno a4=new Alumno("adrian",7);
    Alumno a5=new Alumno("alberto",7);
    Alumno a6=new Alumno("adrian",8);
    TreeSet<Alumno> tre=new TreeSet<Alumno>();
    tre.add(a1);
    tre.add(a2);
    tre.add(a3);
    tre.add(a4);
    tre.add(a5);
    tre.add(a6);
    for (Alumno alumno : tre) {
            System.out.print(alumno+" ");
    }
}//sale: adrian-7.0 adrian-8.0 alberto-6.0 alberto-7.0
```

Otra posibilidad es utilizar un objeto **Comparator<E>**. Esta es otra interfaz que define el método **compare** al que se le pasan los dos objetos a comparar y cuyo resultado es como el de **compareTo** (0 si son iguales, positivo si el primero es mayor y negativo si el segundo es mayor).

Para definir un comparador de esta forma hay que crear una clase que implemente esta interfaz y definir el método **compare**; después crear un objeto de ese tipo y usarle en la construcción del árbol. Ejemplo (comparable al anterior):

```
public class ComparadorAlumnos implements
Comparator<Alumno>{
   public int compare(Alumno o1, Alumno o2) {
          int comparacion=o1.nombre.compareToIgnoreCase(o2.nombre);
          if(comparacion==0){
                return (int)(o1.nota-o2.nota);
          }
          else return comparacion;
}
```

Después al construir la lista se usa el comparador como parámetro en el constructor:

```
TreeSet<Alumno> tre=
          new TreeSet<Alumno>(new ComparadorAlumnos());
```

Normalmente se usa más la interfaz **Comparable**, sin embargo los comparadores de tipo **Comparator** permite ordenar de diferentes formas, por eso en la práctica se utilizan mucho. En el caso de definir la lista con un objeto **Comparator**, esa será la forma prioritaria para ordenar la lista, por encima del método **compareTo** de la interfaz **Comparable**.

Nota: El método **sort** de la clase **Arrays** también admite indicar un comparador para saber de qué forma deseamos ordenar el array.

(9.8) mapas

(9.8.1) introducción a los mapas

Las colecciones de tipo Set tienen el inconveniente de tener que almacenar una copia exacta del elemento a buscar. Sin embargo en la práctica es habitual que haya datos que se consideran clave, es decir que identifican a cada objeto (el dni de las personas por ejemplo) de tal manera que se buscan los datos en base a esa clave y por otro lado se almacenan los datos normales.

Los mapas permiten definir colecciones de elementos que poseen pares de datos clave-valor. Esto se utiliza para localizar valores en función de la clave que poseen. Son muy interesantes y rápidos.

(9.8.2) interfaz Map<K,V>

Es la raíz de todas las clases capaces de implementar mapas. Hasta la versión 1.5, los mapas eran colecciones de pares clave, valor donde tanto la clave como el valor eran de tipo **Object**. Desde la versión 1.5 esta interfaz tiene dos genéricos: **K** para el tipo de datos de la clave y **V** para el tipo de los valores.

Esta interfaz no deriva de **Collection** por lo que no usa iteradores ni ninguno de los métodos vistos anteriormente. La razón es que la obtención, búsqueda y borrado de elementos se hace de manera muy distinta. Los mapas no permiten insertar objetos nulos (provocan excepciones de tipo **NullPointerException**). **Map** define estos métodos:

Método	uso
V **get**(K clave)	Devuelve el objeto que posee la clave indicada
V **put**(Object clave, V valor)	Coloca el par clave-valor en el mapa (asociando la clave a dicho valor). Si la clave ya existiera, sobrescribe el anterior valor y devuelve el objeto antiguo. Si esa clave no aparecía en la lista, devuelve **null**
V **remove**(Object clave)	Elimina de la lista el valor asociado a esa clave. Devuelve el valor que tuviera asociado esa clave o **null** si esa clave no existe en el mapa.
boolean **containsKey**(Object clave)	Indica si el mapa posee la clave señalada
boolean **containsValue**(Object valor)	Indica si el mapa posee el valor señalado
void **putAll**(Map<?extends K, extends V> mapa)	Añade todo el mapa indicado, al mapa actual
Set<K> **keySet**()	Obtiene un objeto *Set* creado a partir de las claves del mapa
Collection<V> **values**()	Obtiene la colección de valores del mapa, permite utilizar el **HashMap** como si fuera una lista normal al estilo de la clase **Collection** (por lo tanto se permite recorrer cada elemento de la lista con un iterador)
int **size**()	Devuelve el número de pares clave-valor del mapa
Set<Map.Entry <K,V>> **entrySet**()	Devuelve una lista formada por objetos **Map.Entry**
void **clear**()	Elimina todos los objetos del mapa

Las operaciones fundamentales son **get**, **put** y **remove**. El conjunto de claves no puede repetir la clave. Hay que tener en cuenta que las claves se almacenan en una tabla hash (es decir es una estructura de tipo **Set**) por lo que para detectar si una clave está repetida, la clase a la que pertenecen las claves del mapa deben definir (si no lo está ya) adecuadamentelos métodos **hashCode** y **equals**.

(9.8.3) interfaz Map.Entry<K,V>

La interfaz Map.Entry se define de forma interna a la interfaz Map y representa un objeto de par clave/valor. Es decir mediante esta interfaz podemos trabajar con una entrada del mapa. Tiene estos métodos:

método	uso
K getKey()	Obtiene la clave del elemento actual Map.Entry
V getValue()	Obtiene el valor
V setValue(V valor)	Cambia el valor y devuelve el valor anterior del objeto actual
boolean equals(Object obj)	Devuelve verdadero si el objeto es un Map.Entry cuyos pares clave-valor son iguales que los del Map.Entry actual

(9.8.4) clases de mapas

clase HashMap

Es la clase más utilizada para implementar mapas. Todo lo comentado con la interfaz Map funciona en la clase HashMap, no añade ningún método o forma de funcionar particular.
Ejemplo:

```
public static void main(String[] args) {
    Alumno a1=new Alumno("alberto",7);
    Alumno a2=new Alumno("mateo",8.5);
    Alumno a3=new Alumno("julián",7.2);
    Alumno a4=new Alumno("adrian",8);
    Alumno a5=new Alumno("alberto",7);
    Alumno a6=new Alumno("adrian",8);
    HashMap<Integer,Alumno> l=new HashMap<Integer,Alumno>();
    l.put(1,a1);
    l.put(2,a2);
    l.put(3,a3);
    l.put(4,a4);
    l.put(5,a5);
    l.put(6,a6);
    System.out.println(l.get(4));
    l.remove(4);
    System.out.println(l);
}
/*Sale:
    adrian-8.0
    {1=alberto-7.0, 2=mateo-8.5, 3=julián-7.2, 5=alberto-7.0, 6=adrian-8.0}
*/
```

Esta clase usa un mapa de dispersión en la tabla Hash que permite que los tiempos de respuesta de **get** y **put** se mantengan aun con muchísimos datos almacenados en el mapa.

Al igual que ocurre con la clase **HashSet**, el orden de entrada no está asegurado. Constructores

constructor	uso
HashMap()	Crea un mapa con una tabla Hash de tamaño 16 y una carga de 0,75
HashMap(Map<?extends K, ?extends V> m)	Crea un mapa colocando los elementos de otro
HashMap(int capacidad)	Crea un mapa con la capacidad indicada y una carga de 0,75
HashMap(int capacidad, float carga)	Crea un mapa con la capacidad y carga máxima indicada

clase LinkedHashMap<K,V>

Es derivada de la anterior, y la única diferencia es que esta clase sí respeta el orden en el que los objetos del mapa fueron insertados. También permite que el orden de los elementos se refiera al último acceso realizado en ellos. Los más recientemente accedidos aparecerán primero. Constructores:

constructor	uso
LinkedHashMap()	Crea un mapa con una tabla Hash de tamaño 16 y una carga de 0,75
LinkedHashMap(Map<?extends K, ?extends V> m)	Crea un mapa colocando los elementos de otro
LinkedHashMap(int capacidad)	Crea un mapa con la capacidad indicada y una carga de 0,75
LinkedHashMap(int capacidad, float carga)	Crea un mapa con la capacidad y carga máxima indicada
LinkedHashMap(int capacidad, float carga, boolean orden)	Crea un mapa con la capacidad y carga máxima indicada. Además si el último parámetro (*orden*) es verdadero el orden se realiza según se insertaron en la lista, de otro modo el orden es por el último acceso.

mapas ordenados. clase TreeMap

Se trata de una estructura de tipo árbol binario, que permite que los elementos del mapa se ordenan en sentido ascendente según la clave. En ese sentido es una clase muy parecida a **TreeSet**.

TreeMap implementa la interfaz **SortedMap** que, a su vez, es heredera de **Map**, por lo que todo lo dicho sobre los mapas funciona con las colecciones de tipo **TreeMap**.

Lo que aportan de nuevo es que los datos en el mapa se ordenan según la clave. Sólo eso y para ordenarlos (al igual que ocurre con la clase **TreeSet**, página 30) la clase de las claves tiene que implementar la interfaz

Comparable o bien durante la creación del TreeMap indicar un objeto Comparator.

Constructores:

constructor	uso
TreeMap()	Crea un mapa que utiliza el orden natural establecido en las claves
TreeMap(Comparator<? super K> m)	Crea un mapa colocando los elementos de otro
TreeMap(Map<?extends K, ?extends V> m)	Crea un mapa a partir de los elementos del mapa indicado, se usará el orden natural de sus claves
TreeMap(SortedSet<K, ?extends V> m)	Crea un mapa usando los elementos y orden de otro mapa

mapas hash débiles. clase WeakHashMap <V,K>

A veces ocurre que cuando se crea un mapa, hay objetos del mismo que no se usan. Es decir claves que nunca se utilizan. Por ello a veces interesaría que esos objetos se eliminaran.

La clase WeakHashMap realiza esa acción, elimina los elementos que no se usan de un mapa y hace que actúe sobre ellos el recolector de basura. Por eso es un mapa de claves débiles. El funcionamiento consiste en detectar las claves que no se han usado y según van apareciendo más, las que llevan más tiempo sin usarse acaban desapareciendo.

Evidentemente este juego es peligroso ya que muchas veces podemos pasar mucho tiempo sin usar una clave y luego necesitarla, por ello hay que usarla sólo con datos de los que estamos seguros que su falta de uso prolongada requiere su eliminación.

(9.9) la clase Collections

Hay una clase llamada Collections (no confundir con la interfaz Collection) que contiene numerosos métodos estáticos para usar con todo tipo de colecciones.

método	uso
static <T>boolean addAll(Collection<? super T> c, T ...elementos)	Añade la lista de elementos de tipo *T* (que irán separados por comas) a la colección indicada por *c*. Java 1.5
static <T> int binarySearch(List<? extends T> l, T valor, Compararator<? super T> c)	Busca de forma binaria el objeto en la lista que deberá estar ordenada según el comparador indicado.
static <T> int binarySearch(List<? extends Comparable <? super T>> l, T valor)	Busca de forma binaria el objeto en la lista que deberá estar ordenada (por ello los elementos de la lista deben de ser de una clase que implemente la interfaz Comparable)

método	uso
static <E> Collection<E> checkedCollection(Collection<E> c, Class <E> tipo)	Devuelve una vista de tipo seguro en el tiempo de ejecución de una colección. Un intento de añadir un elemento de clase incompatible provoca la excepción ClassCastException. Java 1.5
static <E> List<E> checkedList(List<E> *l*, Class <E> tipo)	Igual que la anterior pero orientado a listas doblemente enlazadas. Java 1.5
static <K,V> Map<K,V> checkedMap(Map<K,V> m, Class <K> tipoClave, Class <V> tipoValor)	Funciona como las anteriores pero pensada para mapas. Java 1.5
static <K,V> SortedMap<K,V> checkedSortedMap(SortedMap<K,V> *m*, Class <K> tipoClave, Class <V> tipoValor)	Para mapas ordenados. Java 1.5
static <K,V> SortedMap<K,V> checkedSortedSet(SortedSet<E> s, Class <E> tipo)	Para hash ordenados. Java 1.5
static <T>void copy(List<? super T> lista1, List<? super T> lista2)	Copia los elementos de la segunda lista en la primera. Esa primera tiene que ser tan larga como la segunda, ya que los elementos copiados reemplazan a los originales. Si no es suficientemente grande ocurre una excepción de tipo IndexOutOfBoundsException.
static boolean disjoint(Collection<?> c1, Collection<?> c2)	Devuelve verdadero si ambas colecciones contienen elementos distintos. Java 1.5
static <T>List<T> emptyList()	Devuelve un objeto del tipo List indicado vacío.
static <K,V>Map<K,V> emptyMap()	Devuelve un objeto del tipo Map indicado vacío.
static <T>Set<T> emptySet()	Devuelve un objeto del tipo Set indicado vacío.
static <T>Enumeration<T> enumeration(Collection<T> c)	Obtiene una enumeración a partir de la colección.
static <T>void fill(List<? extends T> lista, T obj)	Reemplaza los elementos de la lista con el objeto indicado. Si la lista tenía 10 elementos, ahora los diez serán una copia del objeto.
static int frequency(Collection<?> c, Object obj)	Cuenta el número de veces que aparece el objeto indicado en la colección. Java 1.5

método	uso
static int **indexOf**(Lista<?> lista, Lista<?> sublista)	Busca todos los elementos de la *sublista* en la lista indicada. Devuelve la posición en la que empieza la sublista. Si no hay esa sublista, devuelve -1
static int **lastIndexOf**(Lista<?> lista, Lista<?> sublista)	Idéntica a la anterior pero empieza a buscar desde el final de la lista, por lo que devuelve la posición inicial en la comienza la última vez que aparece la sublista en la lista (ó -1 si no se encuentra)
static <T> ArrayList<T> **list**(Enumeration<T> enum)	Obtiene una lista de tipo **ArrayList** a partir de la enumeración indicada.
static <T> T **max**(Collection<? extends T> c, Comparator<? super T> comp)	Devuelve el mayor valor de la colección, comparando sus elementos según el comparador indicado.
static <T extends Object & Comparable <? super T>> T max **max**(Collection<? extends T> c)	Devuelve el mayor valor de la colección, que debe implementar la interfaz **Comparable** e indicar en el método **comparteTo** la forma de ordenar
static <T> T **min**(Collection<? extends T> c, Comparator<? super T> comp)	Devuelve el menor valor de la colección, comparando sus elementos según el comparador indicado.
static <T extends Object & Comparable <? super T>> T max **min**(Collection<? extends T> c)	Devuelve el menor valor de la colección, que debe implementar la interfaz **Comparable** e indicar en el método **comparteTo** la forma de ordenar
static <T> List<T> **nCopies**(int num, T obj)	Obtiene una lista formada por tantas copias como indique el parámetro *num*, del objeto indicado.
static <T>boolean **replaceAll**(List< T> lista, T antiguo, T nuevo)	Reemplaza en la lista todas las apariciones del elemento antiguo por el nuevo. Devuelve **false** si no pudo realizar ni un reemplazo (por que el objeto antiguo no está en la lista)
static <T>boolean **reverse**(List< T> lista)	Invierte el orden de la lista
static <T>Comparator<T> **reverseOrder**(Comparator<T> comparador)	Obtiene un comparador que invierte el orden en el que ordena el comparador que recibe como parámetro. Java 1.5
static <T>Comparator<T> **reverseOrder**(Comparator<T> comparador)	Obtiene un comparador que invierte el orden en el que ordena normalmente la clase T (que deberá implementar la interfaz **Comparable**). Por ejemplo: `Arrays.sort(a, Collections.reverseOrder())` ordenaría el array *a* al revés del orden normal en el que se solería ordenar.

método	uso
static void **rotate**(List<T> lista, int n)	Rota la lista el número de elementos indicados por *n*. Si la lista *l* es *[1,2,3,4]*, esta instrucción `Collections.rotate(l,1)` dejaría la lista con los valores *[2,3,4,1]* El número *n* puede ser negativo y entonces la rotación es a la izquierda
static void **shuffle**(List<T> lista)	Mezcla aleatoriamente los elementos de la lista.
static void **shuffle**(List<T> lista, Random r)	Mezcla aleatoriamente la lista utilizando como semilla de rotación el objeto *r*.
static <T> Set <T> **singleton**(T obj)	Devuelve *obj* como una lista de tipo **Set** inmutable y serializable.
static <T> List <T> **singletonList**(T obj)	Devuelve *obj* como una lista de tipo **List** inmutable y serializable. Java 1.3
static <K,V> Map <K,V> **singletonMap**(K clave, V valor)	Devuelve el elemento de clave y valor indicados como un mapa inmutable y serializable. Java 1.3
static <T extends Comparable<? super T> void **sort**(List<T> lista)	Ordena la lista según el orden natural de los elementos (cuya clase debe implementar la interfaz **Comparable**) de la misma.
static <T> void **sort**(List<T> lista, <Comparator <? super t> comp)	Ordena la lista en base al comparador indicado. Los elementos iguales no se reordenan entre sí.
static <T> void **swap**(List<T> lista, int i1, int i2)	Intercambia en la lista los elementos cuyos índices son los indicados. La lista debe poseer elementos en dichos índices son pena de provocar la excepción: **IndexOutOfBoundsException**
static <T> Collection <T> **synchronizedCollection**(Collection <T> c)	Obtiene una colección sincronizada para hilos seguros a partir de la colección indicada
static <T> List <T> **synchronizedList**(List <T> lista)	Obtiene una lista sincronizada para hilos seguros a partir de la lista indicada
static <K,V> SortedMap <K,V> **synchronizedMap**(Map <K,V> mapa)	Obtiene un mapa sincronizado para hilos seguros a partir del mapa indicado
static <T> Set <T> **synchronizedSet**(Set <T> set)	Obtiene una lista sin duplicados sincronizada para hilos seguros a partir de la lista indicada
static <K,V> SortedMap <K,V> **synchronizedSortedMap**(SortedMap <K,V> mapa)	Obtiene un mapa ordenado sincronizado para hilos seguros a partir del mapa indicado

método	uso
static <T> SortedSet <T> synchronizedSortedSet(Set <T> set)	Obtiene una lista ordenada sin duplicados sincronizada para hilos seguros a partir de la lista indicada
static <T>Collection<T> unmodifiableCollection(Collection<? extends T> c)	Obtiene una colección de sólo lectura a partir de la colección indicada
static <T>List<T> unmodifiableList(List<? extends T> lista)	Obtiene una lista de sólo lectura a partir de la lista indicada
static <K,V>Map<K,V> unmodifiableMap(Map<? extends K, ? extends V> mapa)	Obtiene un mapa de sólo lectura a partir del indicado
static <T>Set<T> unmodifiableSet(Set<? extends T> lista)	Obtiene una lista de valores únicos, de sólo lectura a partir de la lista indicada
static <K,V>SortedMap<K,V> unmodifiableSortedMap(SortedMap<? extends K, ? extends V> mapa)	Obtiene un mapa ordenado de sólo lectura a partir del indicado
static <T>Set<T> unmodifiableSortedSet(Set<? extends T> lista)	Obtiene una lista de valores únicos ordenados, de sólo lectura a partir de la lista indicada