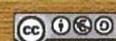


Unidad 8: Control de Errores Mediante Excepciones

Fundamentos de Programación. 1º de ASI



Esta obra está bajo una licencia de Creative Commons.

Autor: Jorge Sánchez Asenjo (año 2010) <http://www.jorgesanchez.net>
e-mail:info@jorgesanchez.net

Esta obra está bajo una licencia de Reconocimiento-NoComercial-CompartirIgual de Creative Commons

Para ver una copia de esta licencia, visite:

<http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es>

o envíe una carta a:

Creative Commons, 559 Nathan Abbot



Reconocimiento-NoComercial-CompartirIgual 2.5 España

Usted es libre de:



copiar, distribuir y comunicar públicamente la obra

hacer obras derivadas

Bajo las condiciones siguientes:



Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).

No comercial. No puede utilizar esta obra para fines comerciales.

Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor
- Apart from the remix rights granted under this license, nothing in this license impairs or restricts the author's moral rights.

Advertencia

Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.
Esto es un resumen legible por humanos del texto legal (la licencia completa) disponible en los idiomas siguientes:

Catalán Castellano Euskera Gallego

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

(8) control de errores

esquema de la unidad

| | |
|--------------------------------------|----|
| (8.1) introducción a las excepciones | 5 |
| (8.2) try y catch | 6 |
| (8.3) manejo de excepciones | 9 |
| (8.4) métodos de la clase Exception | 10 |
| (8.5) throws | 10 |
| (8.6) throw | 11 |
| (8.7) finally | 12 |

(8.1) introducción a las excepciones

Uno de los problemas más importantes al escribir aplicaciones es el tratamiento de los errores. Los errores detienen la ejecución del programa e impiden su desarrollo normal y, lo peor, además provocan que el usuario esté desinformado. Toda programadora o programador tiene que reconocer las situaciones que pueden provocar el fin de la ejecución normal del programa por un error no controlado.

Dicho de otra forma, todos los posibles errores en un programa deben de estar controlado. A veces es imposible evitarlos (por ejemplo no hay papel en la impresora, o falla el disco duro), pero sí reaccionar de forma lógica para que el usuario o usuaria reconozca lo que está ocurriendo.

Java nos echa una mano para ello a través de las **excepciones**. Se denomina excepción a un hecho que podría provocar la detención del programa; es decir una condición de error en tiempo de ejecución pero que puede ser controlable (a través de los mecanismos adecuados). En java sin embargo se denomina **error** a una condición de error incontrolable (ejemplos son el error que ocurre cuando no se dispone de más memoria o errores de sintaxis).

Ejemplos de excepciones son:

- ◆ El archivo que queremos abrir no existe
- ◆ Falla la conexión a una red

Ejemplo:

```
try {
    readFile("arch");
    ...
}

catch(FileNotFoundException e) {
    //archivo no encontrado
    ...
}

catch (IOException e) {
    ...
}
```

Si en las instrucciones del bloque **try** hay un error causado por no encontrar el archivo se producirá una excepción de tipo **FileNotFoundException** y será manejada por el catch correspondiente, si se produce del otro tipo (**IOException**) se maneja por el siguiente, si se produce de otro tipo, el programa se detendrá.

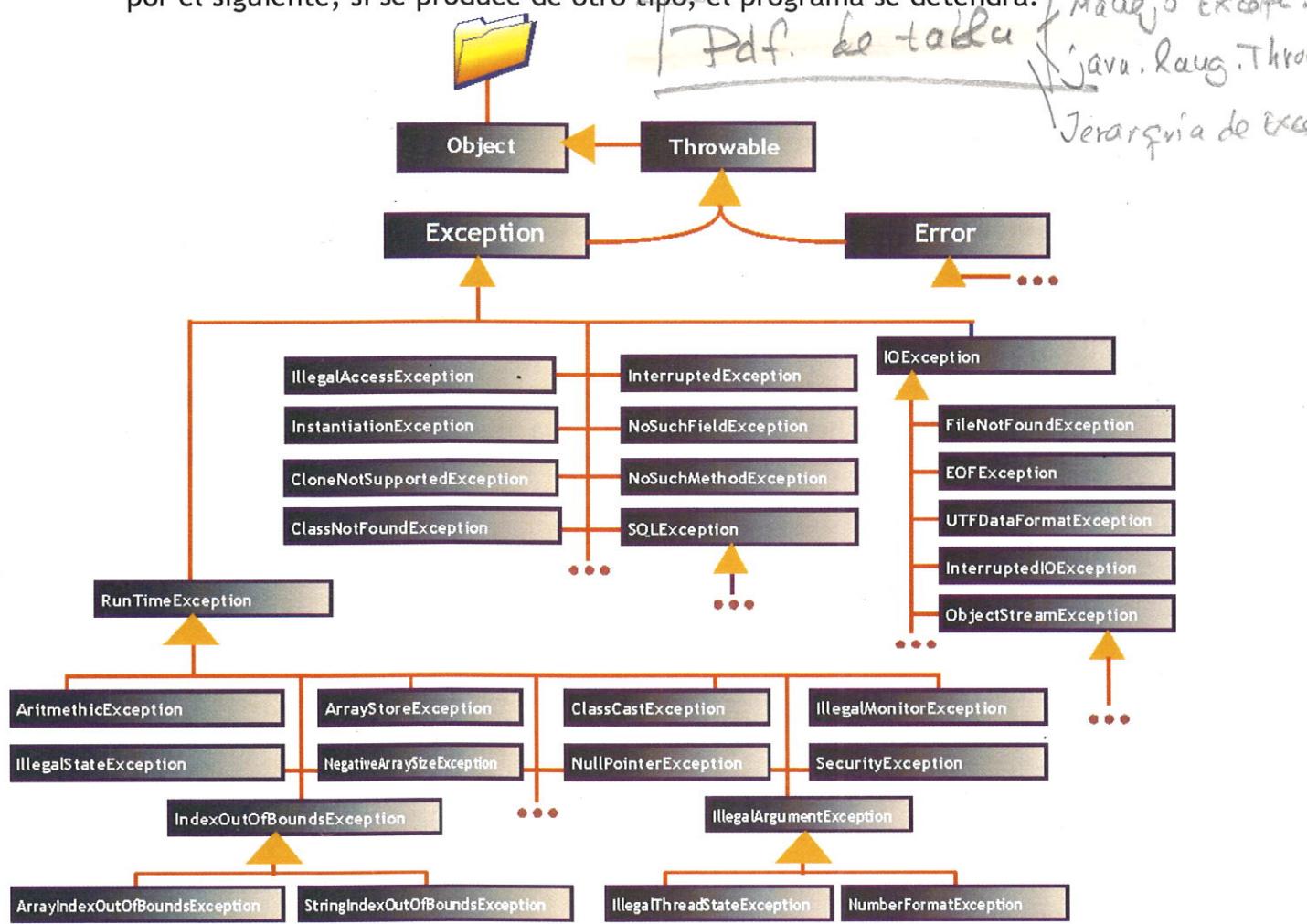


Ilustración 8-1, Diagrama de algunas de las clases fundamentales para el control de excepciones

(8.3) manejo de excepciones

Siempre se debe controlar una excepción, de otra forma nuestro software está a merced de los fallos. En la programación de aplicaciones en general siempre ha habido dos formas de manejar la excepción:

- ◆ **Interrupción.** En este caso se asume que el programa ha encontrado un error irrecuperable. La operación que dio lugar a la excepción se anula y se entiende que no hay manera de regresar al código que provocó la excepción. Es decir, la operación que dio pie al error, se anula.
- ◆ **Reanudación.** Se puede manejar el error y regresar de nuevo al código que provocó el error.

La filosofía de Java es del tipo interrupción, pero se puede intentar emular la reanudación encerrando el bloque **try** en un bucle que se repetirá hasta que el error deje de existir. Ejemplo:

```
boolean indiceNoValido=true;
int i; //Entero que tomará los aleatorios de 0 a 9
String texto[]={ "Uno", "Dos", "Tres", "Cuatro", "Cinco"};
while(indiceNoValido){
    try{
        i=(int)(Math.round(Math.random()*9));
        System.out.println(texto[i]);
        indiceNoValido=false;
    }catch(ArrayIndexOutOfBoundsException exc){
        System.out.println("Fallo en el índice");
    }
}
```

En el código anterior, el índice **i** calcula un número aleatorio del 0 al 9 y con ese número el código accede al array **texto** que sólo contiene 5 elementos. Esto producirá continuamente (ya que el array es mucho más pequeño) una excepción del tipo **ArrayIndexOutOfBoundsException** que es manejada por el **catch** correspondiente.

Para buscar otro intento, el bloque **catch** está dentro de un bucle **while**, que permite otro intento y así hasta que no haya excepción (es decir hasta que el número esté entre 0 y 5), lo que provocará que **indiceNoValido** valga **true** y la salida, al fin, del **while**. Es un tanto enrevesado el código pero valga como ejemplo del funcionamiento de la captura de errores con reintento.

correspondientes. Cuando un método hace uso de **throws**, en el **javadoc** del mismo se documenta de esta forma:

```
/**  
 * Método que abre y prepara el archivo  
 * @param archivo Nombre del archivo a preparar  
 * @throws IOException Si ocurre un error en la apertura  
 * @throws InterruptedException Si hay un error en el  
 * contenido del archivo  
 */  
  
void usarArchivo (String archivo) throws IOException,  
InterruptedException {  
    ...  
}
```

Para utilizar el método:

```
try{  
    ...  
    objeto.usarArchivo("C:\texto.txt");//puede haber excepción  
    ...  
}  
catch(IOException ioe){  
    ...  
}  
catch(InterruptedException ie){  
    ...  
}  
...
```

(8.6) throw

Esta instrucción nos permite provocar a nosotros una excepción (o lo que es lo mismo, crear artificialmente nosotros las excepciones). Ante:

throw new Exception();

El flujo del programa se dirigirá a la instrucción **try...catch** más cercana. Se pueden utilizar constructores en esta llamada (el formato de los constructores depende de la clase que se utilice):

throw new Exception("Error grave, grave");

Eso construye una excepción con el mensaje indicado.

Ejercicio42

1. Dada la siguiente clase Alumno, con sus atributos y sus métodos:

```
public class Alumno {  
    private String nombre;  
    private String apellido;  
    private Integer edad;  
  
    public Alumno(String nombre, String apellido, Integer edad) throws InvalidYearException {  
        this.nombre = nombre;  
        this.apellido = apellido;  
        if (edad <= 0){  
            throw new InvalidYearException ("La edad debe ser mayor a cero");  
        }else{  
            this.edad = edad;  
        }  
    }  
    public String getNombre() {  
        return nombre;  
    }  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
    public String getApellido() {  
        return apellido;  
    }  
    public void setApellido(String apellido) {  
        this.apellido = apellido;  
    }  
    public Integer getEdad() {  
        return edad;  
    }  
}
```

2. Implementa la clase NewYearException.

3. Implementa un método main() donde se capture esta excepción.
Pruébala intentando instanciar un alumno con una edad negativa.

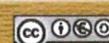
Unidad 9:

Colecciones de

datos

Fundamentos de Programación.

1º de ASI



Esta obra está bajo una licencia de Creative Commons.

Autor: Jorge Sánchez Asenjo (año 2010) <http://www.jorgesanchez.net>
e-mail:info@jorgesanchez.net

Esta obra está bajo una licencia de Reconocimiento-NoComercial-CompartirIgual de Creative Commons

Para ver una copia de esta licencia, visite:

<http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es>

o envíe una carta a:

Creative Commons, 559 Nathan Abbot



Reconocimiento-NoComercial-CompartirIgual 2.5 España

Usted es libre de:



- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

Bajo las condiciones siguientes:



Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).

No comercial. No puede utilizar esta obra para fines comerciales.

Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor
- Apart from the remix rights granted under this license, nothing in this license impairs or restricts the author's moral rights.

Advertencia

Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.
Esto es un resumen legible por humanos del texto legal (la licencia completa) disponible en los idiomas siguientes:
Catalán Castellano Euskera Gallego

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 | 6 |
| (9.2) jerarquía de clases Java para implementar colecciones | 8 |
| (9.3) interfaz Collection, listas simplemente enlazadas | 9 |
| (9.3.1) iteradores | 10 |
| (9.3.2) for each para colecciones | 11 |
| (9.4) interfaz List, listas doblemente enlazadas | 11 |
| (9.4.1) interfaz List | 11 |
| (9.4.2) ListIterator | 12 |
| (9.4.3) clase ArrayList | 13 |
| (9.4.4) clase LinkedList | 14 |
| (9.5) genéricos de Java | 15 |
| (9.5.1) introducción a los genéricos | 15 |
| (9.5.2) idea general de los genéricos | 15 |
| (9.5.3) genéricos en métodos | 16 |
| (9.5.4) genéricos en las clases | 17 |
| (9.5.5) interfaces con genéricos | 18 |
| (9.5.6) uso de varios genéricos | 18 |
| (9.5.7) limitación de los tipos posibles en un genérico | 19 |
| (9.5.8) tipos comodín | 20 |
| (9.5.9) genéricos en las colecciones | 21 |
| (9.6) colecciones sin duplicados | 25 |
| interfaz Set | 25 |
| clase HashSet | 25 |
| (9.6.1) clase LinkedHashSet | 28 |
| (9.7) árboles, listas ordenadas | 28 |
| (9.7.1) árboles | 28 |
| (9.7.2) interfaz SortedSet<E> | 29 |
| (9.7.3) clase TreeSet<E> | 30 |
| (9.7.4) comparaciones | 30 |
| (9.8) mapas | 31 |
| (9.8.1) introducción a los mapas | 31 |
| (9.8.2) interfaz Map<K,V> | 32 |
| (9.8.3) interfaz Map.Entry<K,V> | 33 |
| (9.8.4) clases de mapas | 33 |
| (9.9) la clase Collections | 35 |

- ◆ **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 implementa siempre mediante colecciones de datos. En este caso la clase Polígono crea internamente (al ser una composición) una lista de Vértices (de los que no puede haber menos de tres)

(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 |
|---|--|
| <code>boolean add(Object o)</code> | 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 |
| <code>boolean remove(Object o)</code> | Elimina al objeto indicado de la colección. |
| <code>int size()</code> | Devuelve el número de objetos almacenados en la colección |
| <code>boolean isEmpty()</code> | Indica si la colección está vacía |
| <code>boolean contains(Object o)</code> | Devuelve true si la colección contiene al objeto indicado <i>o</i> |
| <code>void clear()</code> | Elimina todos los elementos de la colección |
| <code>boolean addAll(Collection otra)</code> | Añade todos los elementos de la colección <i>otra</i> a la colección actual |
| <code>boolean removeAll(Collection otra)</code> | Elimina todos los objetos de la colección actual que estén en la colección <i>otra</i> |
| <code>boolean retainAll(Collection otra)</code> | Elimina todos los elementos de la colección que no estén en la otra |
| <code>boolean containsAll(Collection otra)</code> | Indica si la colección contiene todos los elementos de otra |
| <code>Object[] toArray()</code> | Convierte la colección en un array de objetos. |
| <code>Iterator iterator()</code> | 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 objetos 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).

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 objeto: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 colecciónString es una colección de textos  
for(Object o:colecciónString){  
    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 |

Ejercicio nº. 43

Diseñar una clase Principal cuyo método *main()* realice las siguientes acciones:

1. Crea un ArrayList con 10 referencias a enteros y cuyos valores se han obtenido de forma aleatoria entre 1 y 10.
2. A continuación lo muestra en pantalla.
3. Llama a un método que elimina del ArrayList todas las ocurrencias del entero con valor 5, devolviendo el resultado.
4. Transforma el ArrayList en un array estático de enteros.
5. Muestra en pantalla el array de Enteros.

Nota: Los métodos pueden estar implementados en la misma clase Principal.

referencia al
objeto List

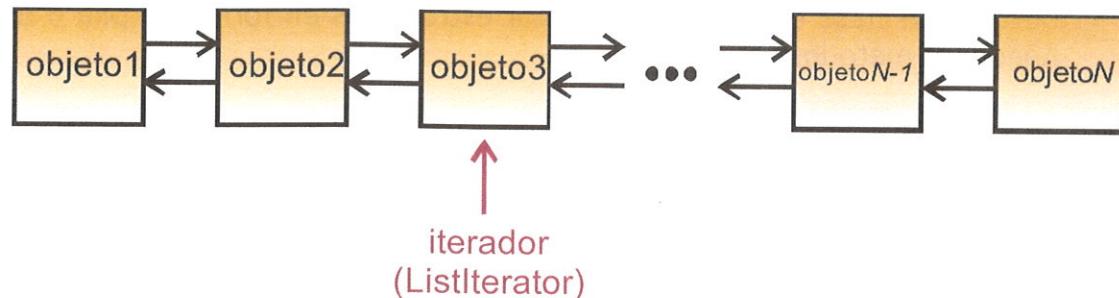


Ilustración 9-4, Representación gráfica de una colección de datos de tipo List. El iterador puede recorrer la lista en cualquier dirección

(9.4.3) clase ArrayList

Implementa la interfaz **List**. Está pensada para crear tanto listas simples 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.

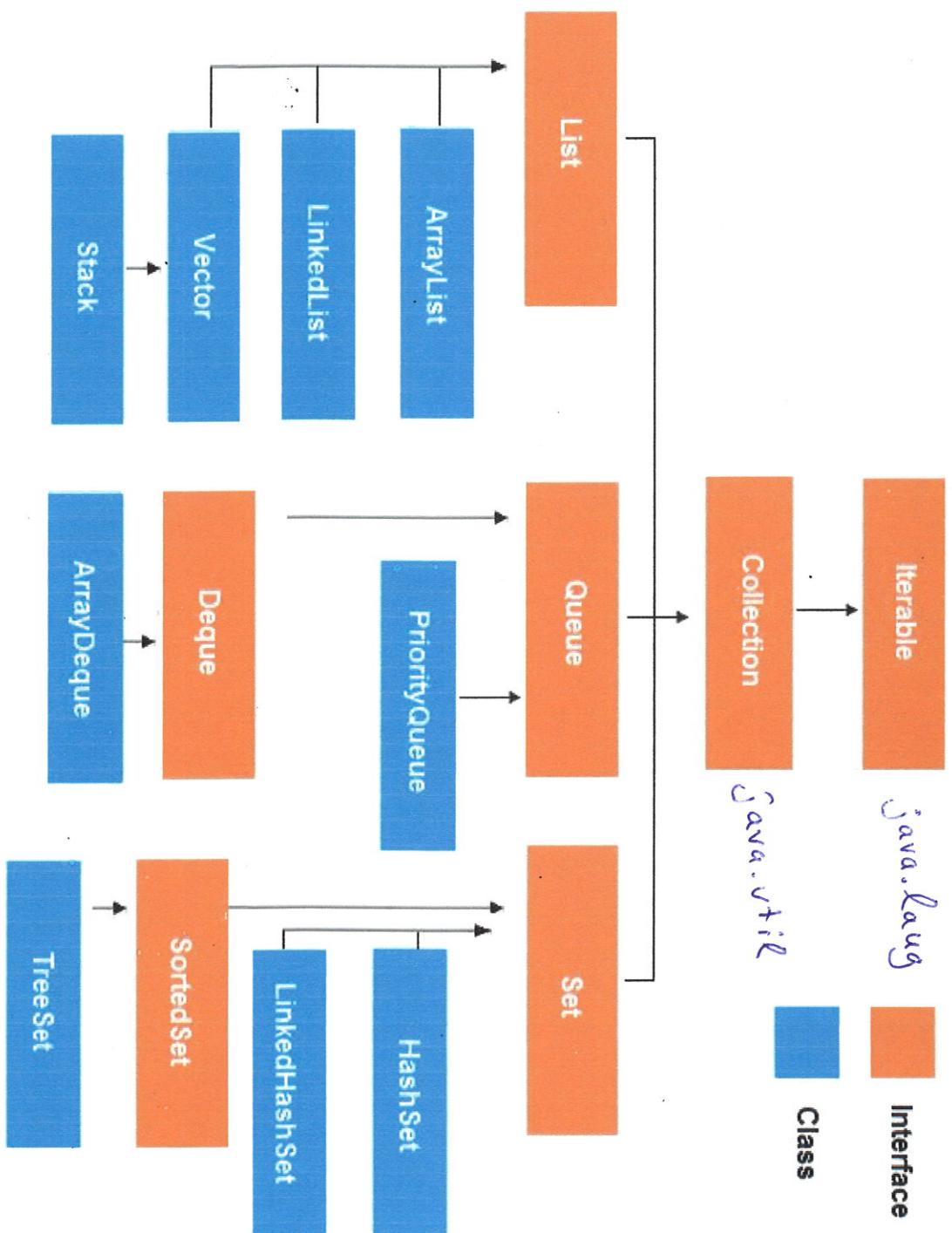
Ejemplo:

```
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
*/
```

Ejercicio uº 43

(13)

-Pdf Adjunto.



Ejercicio nº. 35 44

Utilizando una colección LinkedList:

Diseñar un programa que pida los datos de empleados. Para ello, se hará una pregunta como esta al usuario:

¿Quiere insertar los datos de algún empleado (S/N)?

De cada empleado nos interesa su **nombre y su sueldo**.

Una vez finalizada la inserción de todos los empleados:

- a) Visualizar los datos de todos los empleados en el mismo orden que se insertaron.
- b) El programa pedirá el nombre de un empleado con el fin de borrarlo de la lista. Esta operación se realizará varias veces hasta que el usuario indique que no quiere borrar ninguno más.
- c) El programa pedirá una posición para visualizar los datos del empleado que está en esa posición. Ten en cuenta que el usuario dará las posiciones de 1 en adelante y puede que se indique una posición que no exista. Esta operación se realizará varias veces hasta que el usuario indique que no quiere visualizar ninguno más.
- d) El programa permitirá al usuario modificar los datos de un empleado concreto, indicando para ello la posición del empleado que quiere cambiar. Ten en cuenta lo anterior (posiciones de 1 en adelante). Esta operación se realizará varias veces hasta que el usuario indique que no quiere modificar ninguno más.
- e) Por último, el programa dará la opción de insertar nuevos empleados en la posición que indique el usuario del programa. Ten en cuenta lo anterior (posiciones de 1 en adelante). Esta operación se realizará varias veces hasta que el usuario indique que no quiere insertar ninguno más.
- f) Después de cada operación se visualizará la lista para ver su estado actual.

Ejercicio45

Crea una clase denominada Aves con 4 atributos:

- tipoDeAve
- numeroDeAves
- numeroDeMachos
- numeroDeHembras

Crea una clase GruposDeAves que implemente la interface Iterable. Crea varios objetos de tipo Aves y añádelos a una colección GruposDeAves utilizando un iterador.

Crea la clase **Ejercicio45** que represente los datos de los objetos presentes en la colección GruposDeAves.

El resultado a mostrar por pantalla deberá ser similar al siguiente:

| TIPO | MACHOS | HEMBRAS | TOTAL |
|----------|--------|---------|-------|
| Aguilas | 10 | 25 | 35 |
| Buitres | 55 | 45 | 100 |
| Halcones | 80 | 25 | 55 |

(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**.

(9.5.4) 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

Ejercicio nº. 37 48

Diseñar la Clase Genérica Operaciones, con la siguiente funcionalidad:

- Calcular el elemento mínimo de un array;
- Calcular el elemento máximo de un array;
- Buscar un Objeto en un array, devolviendo la posición de la primera aparición del Objeto en el array o -1 si no existe.
- Borrar la primera aparición en el array de un objeto dado, devolviendo el objeto eliminado, o null si no existe.

Además:

- Crear la clase TestOperaciones con método main() que use la Clase Operaciones para un array de Integer y para un array de Float

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.

(19)

Ejercicio 4'506

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.indice==this.indice;
}
```

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.indice==this.indice;
}
```

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.

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

métodos de List<E>

| método | uso |
|---|---|
| <code>void add(int índice, E elemento)</code> | Añade el elemento indicado en la posición <code>índice</code> de la lista |
| <code>void remove(int índice)</code> | Elimina el elemento cuya posición en la colección la da el parámetro <code>índice</code> |
| <code>E set(int índice, E elemento)</code> | Sustituye el elemento número <code>índice</code> por uno nuevo. Devuelve además el elemento antiguo |
| <code>E get(int índice)</code> | Obtiene el elemento almacenado en la colección en la posición que indica el índice |
| <code>int indexOf(Object elemento)</code> | Devuelve la posición del elemento. Si no lo encuentra, devuelve -1 |
| <code>int lastIndexOf(Object elemento)</code> | Devuelve la posición del elemento comenzando a buscarle por el final. Si no lo encuentra, devuelve -1 |
| <code>void addAll(int índice, Collection <? extends E> elemento)</code> | Añade todos los elementos de una colección compatible a partir de la posición dada. |
| <code>List<E> subList(int inicio, int fin)</code> | Devuelve una lista del mismo tipo que la actual tomando los elementos que van desde el inicio indicado al fin (sin incluir este último). |
| <code>ListIterator<E> listIterator()</code> | Obtiene el iterador de lista que permite recorrer los elementos de la lista |
| <code>ListIterator<E> listIterator(int índice)</code> | 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 |
|--------------------------------|--|
| <code>E next()</code> | 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: <code>NoSuchElementException</code> (que deriva a su vez de <code>RunTimeException</code>) |
| <code>boolean hasNext()</code> | Indica si hay un elemento siguiente (y así evita la excepción). |
| <code>void remove()</code> | Elimina el último elemento devuelto por <code>next</code> |

(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.

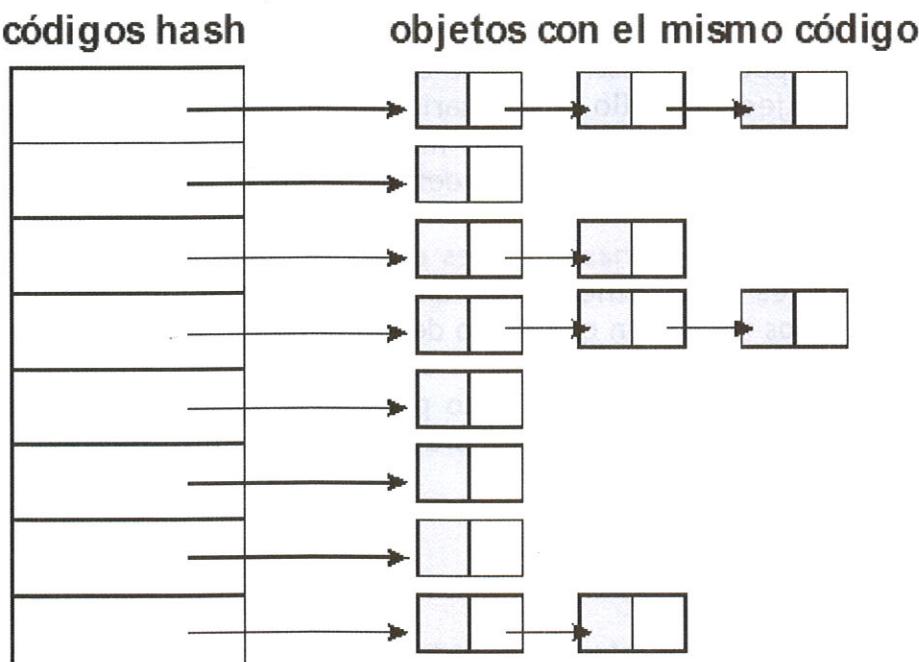


Ilustración 9-6, Ejemplo de tabla hash, varios elementos se asocian con el mismo código. La tabla (un array) mantiene los códigos únicos por un lado y en cada hay una lista de los objetos con dicho código

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.

Ejercicio uº 46

Pdf Adjunto.

1º curso de administración de sistemas informáticos
autor: Jorge Sánchez - www.jorgesanchez.net

```
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.

Ejercicio uº 47

- Crear la clase JugadorSeleccion con tres propiedades: dorsal, nombre, posición.
- Almacenar en un HashMap, objetos de la clase JugadorSeleccion.
- Recuperar la colección con un iterador

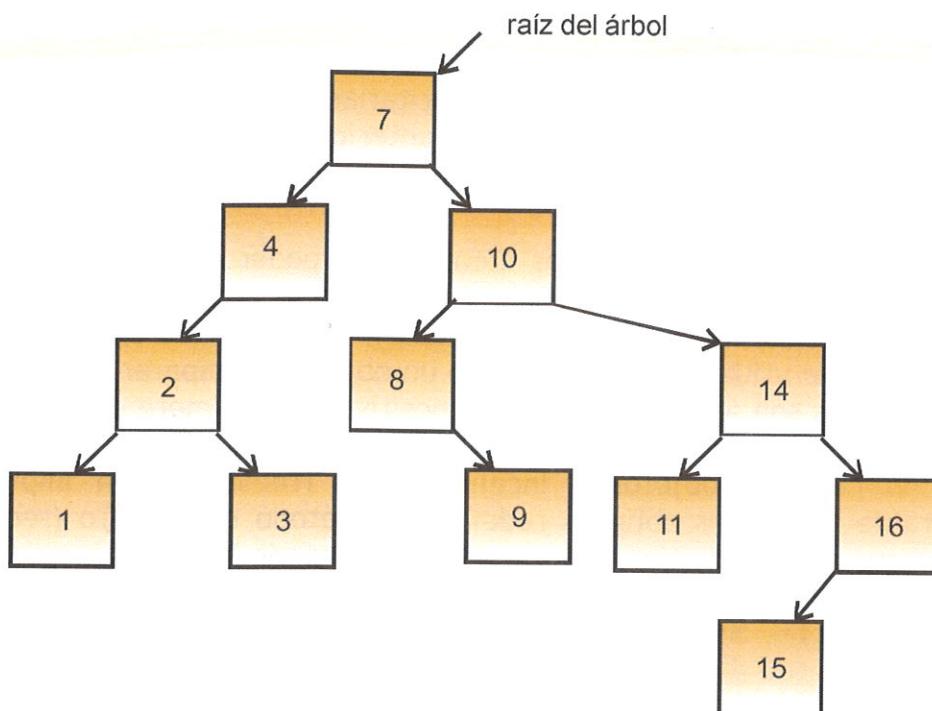


Ilustración 9-7, Representación gráfica del funcionamiento de un árbol binario ordenado.

(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 <i>o</i> . |
| SortedSet tailSet(E o) | Obtiene un SortedSet que contendrá todos los elementos mayores que el objeto <i>o</i> . |
| 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**).

(*)

```
TreeSet<Alumno> tsm = new TreeSet<Alumno>();
ESM = (TreeSet<Alumno>) ls. headset(al4);
```

(**) Implementa el método **compare(Object obj1, Object obj2)** de la interfaz **Comparable**. Alternativamente a **comparableTo(Object obj)** de la interfaz **Comparable**.

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 usarlo 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.

(9.8.3) 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}*/

```

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

Constructores:

| constructor | uso |
|---|---|
| <code>TreeMap()</code> | Crea un mapa que utiliza el orden natural establecido en las claves |
| <code>TreeMap(Comparator<? super K> m)</code> | Crea un mapa colocando los elementos de otro |
| <code>TreeMap(Map<?extends K, ?extends V> m)</code> | Crea un mapa a partir de los elementos del mapa indicado, se usará el orden natural de sus claves |
| <code>TreeMap(SortedSet<K, ?extends V> m)</code> | 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 |
|---|---|
| <code>static <T>boolean addAll(Collection<? super T> c, T ...elementos)</code> | Añade la lista de elementos de tipo <code>T</code> (que irán separados por comas) a la colección indicada por <code>c</code> . Java 1.5 |
| <code>static <T> int binarySearch(List<? extends T> l, T valor, Comparator<? super T> c)</code> | Busca de forma binaria el objeto en la lista que deberá estar ordenada según el comparador indicado. |
| <code>static <T> int binarySearch(List<? extends Comparable <? super T>> l, T valor)</code> | 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 |
|---|--|
| <code>static int indexOf(List<?> lista, Lista<?> sublista)</code> | Busca todos los elementos de la <i>sublista</i> en la lista indicada. Devuelve la posición en la que empieza la sublista. Si no hay esa sublista, devuelve -1 |
| <code>static int lastIndexOf(List<?> lista, Lista<?> sublista)</code> | 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) |
| <code>static <T> ArrayList<T> list(Enumeration<T> enum)</code> | Obtiene una lista de tipo ArrayList a partir de la enumeración indicada. |
| <code>static <T> T max(Collection<? extends T> c, Comparator<? super T> comp)</code> | Devuelve el mayor valor de la colección, comparando sus elementos según el comparador indicado. |
| <code>static <T extends Object & Comparable <? super T>> T max(max(Collection<? extends T> c))</code> | Devuelve el mayor valor de la colección, que debe implementar la interfaz Comparable e indicar en el método comparteTo la forma de ordenar |
| <code>static <T> T min(Collection<? extends T> c, Comparator<? super T> comp)</code> | Devuelve el menor valor de la colección, comparando sus elementos según el comparador indicado. |
| <code>static <T extends Object & Comparable <? super T>> T min(min(Collection<? extends T> c))</code> | Devuelve el menor valor de la colección, que debe implementar la interfaz Comparable e indicar en el método comparteTo la forma de ordenar |
| <code>static <T> List<T> nCopies(int num, T obj)</code> | Obtiene una lista formada por tantas copias como indique el parámetro <i>num</i> , del objeto indicado. |
| <code>static <T> boolean replaceAll(List< T > lista, T antiguo, T nuevo)</code> | 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) |
| <code>static <T> boolean reverse(reverse(List< T > lista))</code> | Invierte el orden de la lista |
| <code>static <T> Comparator<T> reverseOrder(Comparator<T> comparador)</code> | Obtiene un comparador que invierte el orden en el que ordena el comparador que recibe como parámetro. Java 1.5 |
| <code>static <T> Comparator<T> reverseOrder(reverseOrder(Comparator<T> comparador))</code> | 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 <i>a</i> al revés del orden normal en el que se solería ordenar. |

| 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 |

A R B O L E S

Ejercicio uº 48

Pdf Adjunto.
(Ejercicio 48A.pdf)

Ejercicio uº 49

Pdf Adjunto.
(Ejercicio49A.pdf)

Pdf Adjunto Hash Maps (Ejercicio 48
HM.pdf)

Pdf Adjunto LinkedList (Ejercicio 49
C.pdf)

Ejercicio nº48

Implementar estáticamente un Árbol Binario de Búsqueda formado por 6 Integer (comprendidos entre 1 y 20) generados al azar.

Representar la Estructura Pre-Ordenr creada.

Ejemplo:

Entrada: 17,14,8,9,6,11

Salida:

| | | |
|----|---|---|
| 17 | 1 | |
| 14 | 2 | |
| 8 | 4 | 3 |
| 9 | | 5 |
| 6 | | |
| 11 | | |

Ejercicio49

Crear una clase Java (Ejercicio49) que, a través del siguiente Menú de opciones:

- 1.- Insertar nuevo elemento.
- 2.- Buscar un elemento.
- 3.- Averiguar menor y mayor de un elemento dado.
- 4.- Devolver distancia entre el menor y el mayor.
- 5.- Averiguar Nivel de un elemento.
- 6.- Eliminar un elemento.
- 7.- Salir.

Pruebe la funcionalidad de un Árbol Binario de Búsqueda compuesto de objetos de la clase Integer.