







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

| método | uso |
|--|---|
| <code>static <E> Collection<E> checkedCollection(Collection<E> c, Class <E> tipo)</code> | 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 |
| <code>static <E> List<E> checkedList(List<E> l, Class <E> tipo)</code> | Igual que la anterior pero orientado a listas doblemente enlazadas. Java 1.5 |
| <code>static <K,V> Map<K,V> checkedMap(Map<K,V> m, Class <K> tipoClave, Class <V> tipoValor)</code> | Funciona como las anteriores pero pensada para mapas. Java 1.5 |
| <code>static <K,V> SortedMap<K,V> checkedSortedMap(SortedMap<K,V> m, Class <K> tipoClave, Class <V> tipoValor)</code> | Para mapas ordenados. Java 1.5 |
| <code>static <K,V> SortedSet<E> checkedSortedSet(SortedSet<E> s, Class <E> tipo)</code> | Para hash ordenados. Java 1.5 |
| <code>static <T>void copy(List<? super T> lista1, List<? super T> lista2)</code> | Copia los elementos de la segunda lista en la primera. Esta 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 . |
| <code>static boolean disjoint(Collection<?> c1, Collection<?> c2)</code> | Devuelve verdadero si ambas colecciones contienen elementos distintos. Java 1.5 |
| <code>static <T>List<T> emptyList()</code> | Devuelve un objeto del tipo List indicado vacío. |
| <code>static <K,V>Map<K,V> emptyMap()</code> | Devuelve un objeto del tipo Map indicado vacío. |
| <code>static <T>Set<T> emptySet()</code> | Devuelve un objeto del tipo Set indicado vacío. |
| <code>static <T>Enumeration<T> enumeration(Collection<T> c)</code> | Obtiene una enumeración a partir de la colección. |
| <code>static <T>void fill(List<? extends T> lista, T obj)</code> | Reemplaza los elementos de la lista con el objeto indicado. Si la lista tenía 10 elementos, ahora los diez serán una copia del objeto. |
| <code>static int frequency(Collection<?> c, Object obj)</code> | Cuenta el número de veces que aparece el objeto indicado en la colección. Java 1.5 |

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

(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) adecuadamente los métodos **hashCode** y **equals**.

(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+ " ");
    }
}//salida: adrian-7.0 adrian-8.0 alberto-6.0 alberto-7.0
```

(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.

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 llenado 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 factor 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 para 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 lo 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;  
    }  
}
```

métodos de ListIterator<E>

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

métodos de LinkedList<E>

| método | uso |
|---------------------------------|--|
| <code>E getFirst()</code> | Obtiene el primer elemento de la lista |
| <code>E getLast()</code> | Obtiene el último elemento de la lista |
| <code>void addFirst(E o)</code> | Añade el objeto al principio de la lista |
| <code>void addLast(E o)</code> | Añade el objeto al final de la lista |
| <code>void removeFirst()</code> | Borra el primer elemento |
| <code>void removeLast()</code> | 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);
}
```

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 |
|--|--|
| <code>boolean add(E 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(E 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 |
| <code>void clear()</code> | Elimina todos los elementos de la colección |
| <code>boolean addAll(Collection<? extends E> otra)</code> | Añade todos los elementos de la colección <i>otra</i> a la colección actual. Sólo puede añadir objetos de colecciones cuyos elementos sean de tipos compatibles con el actual. |
| <code>boolean removeAll(Collection<? extends E> otra)</code> | Elimina todos los objetos de la colección actual que estén en la colección <i>otra</i> |
| <code>boolean retainAll(Collection<? extends E> otra)</code> | Elimina todos los elementos de la colección que no estén en la otra |
| <code>boolean containsAll(Collection<? extends E> 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><T>T[] toArray(T array)</code> | 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). |

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 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 Comparar <T>{  
    T menor();  
    T mayor();  
}
```

Una clase que implemente la interfaz puede hacerlo así:

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

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

```
public class Clase1 <T> implements Comparar<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`.

(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];  
    }  
}
```

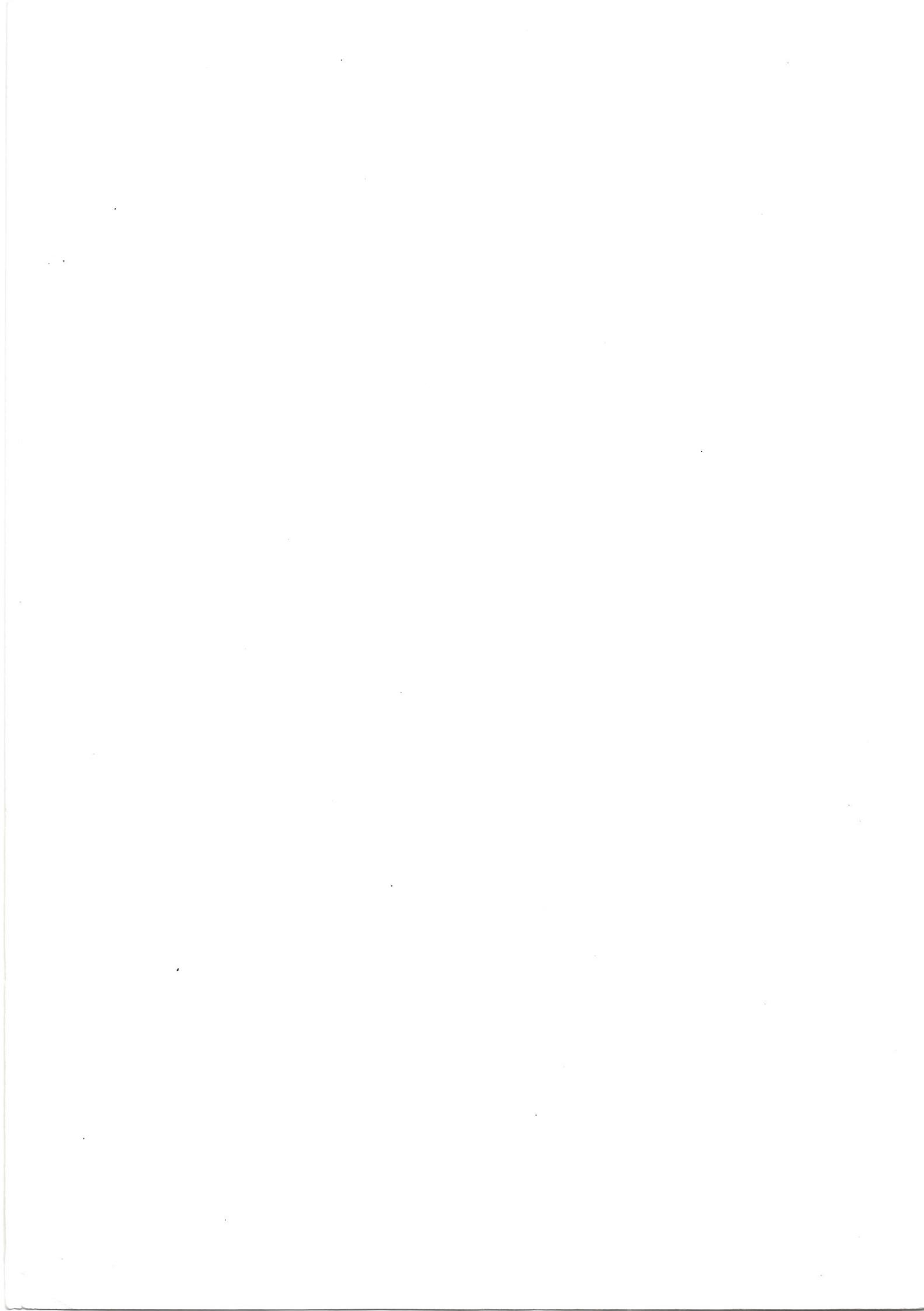
En el ejemplo la letra **T** entre los signos **<** y **>** indica que se va a utilizar un tipo genérico, el hecho de que le siga otra **T** significa que el tipo que devuelve el método **tomarAleatorio** será el mismo que el del parámetro **a**, y se corresponderá al tipo genérico **T**. Después cuando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







(9.4.4) 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 |
|--------------------------------------|--|
| <code>Object getFirst()</code> | Obtiene el primer elemento de la lista |
| <code>Object getLast()</code> | Obtiene el último elemento de la lista |
| <code>void addFirst(Object o)</code> | Añade el objeto al principio de la lista |
| <code>void addLast(Object o)</code> | Añade el objeto al final de la lista |
| <code>void removeFirst()</code> | Borra el primer elemento |
| <code>void removeLast()</code> | Borra el último elemento |

Los métodos están pensados para que las listas creadas mediante objetos **LinkedList** sirven para añadir elementos por la cabeza 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**).

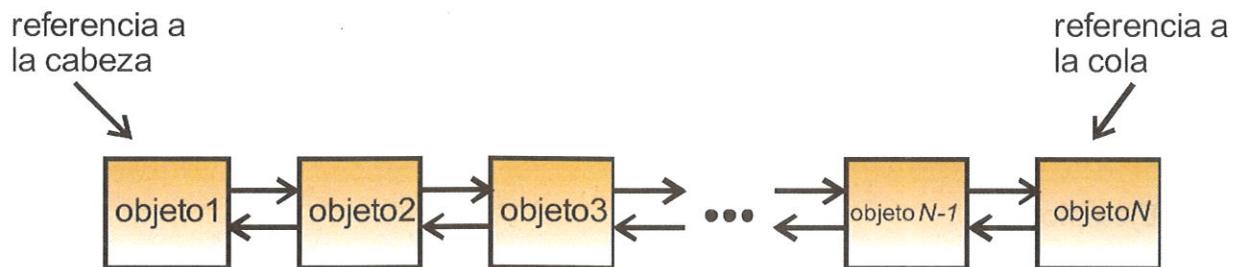


Ilustración 9-5, Representación gráfica de una colección de datos de tipo **LinkedList**. La idea es no utilizar iteradores y manejar sólo los elementos de la cabeza y la cola.

Ejercicio uº 44

- Pdf adjunto.

Ejercicio uº 45

- Pdf adjunto.



| método | uso |
|---|---|
| <code>Object set(int índice, Object elemento)</code> | Sustituye el elemento número <i>índice</i> por uno nuevo. Devuelve además el elemento antiguo |
| <code>Object 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 elemento)</code> | Añade todos los elementos de una colección a una posición dada. |
| <code>ListIterator listIterator()</code> | Obtiene el iterador de lista que permite recorrer los elementos de la lista |
| <code>ListIterator 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. |
| <code>List subList(int desde, int hasta)</code> | Obtiene una lista con los elementos que van de la posición <i>desde</i> a la posición <i>hasta</i> |

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 |
|--|---|
| <code>void add(Object elemento)</code> | Añade el elemento delante de la posición actual del iterador |
| <code>void set(Object elemento)</code> | Sustituye el elemento señalado por el iterador, por el elemento indicado |
| <code>Object previous()</code> | Obtiene el elemento previo al actual. Si no lo hay provoca excepción: <code>NoSuchElementException</code> |
| <code>boolean hasPrevious()</code> | Indica si hay elemento anterior al actualmente señalado por el iterador |
| <code>int nextIndex()</code> | Obtiene el índice del elemento siguiente |
| <code>int previousIndex()</code> | 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 contiene todos los métodos de `Iterator` (como `remove` por ejemplo) ya que los hereda.

(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 **Iterator**).

referencia al
objeto Collection

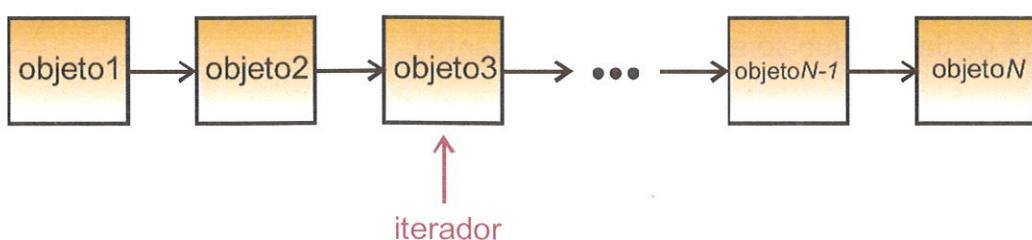


Ilustración 9-3, Representación gráfica de una lista simplemente enlazada (que es lo que implementa la clase **Collection**), el iterador en todo momento señala a un objeto de la lista, pero sólo puede avanzar de izquierda a derecha en la lista

(9.2) jerarquía de clases Java para implementar colecciones

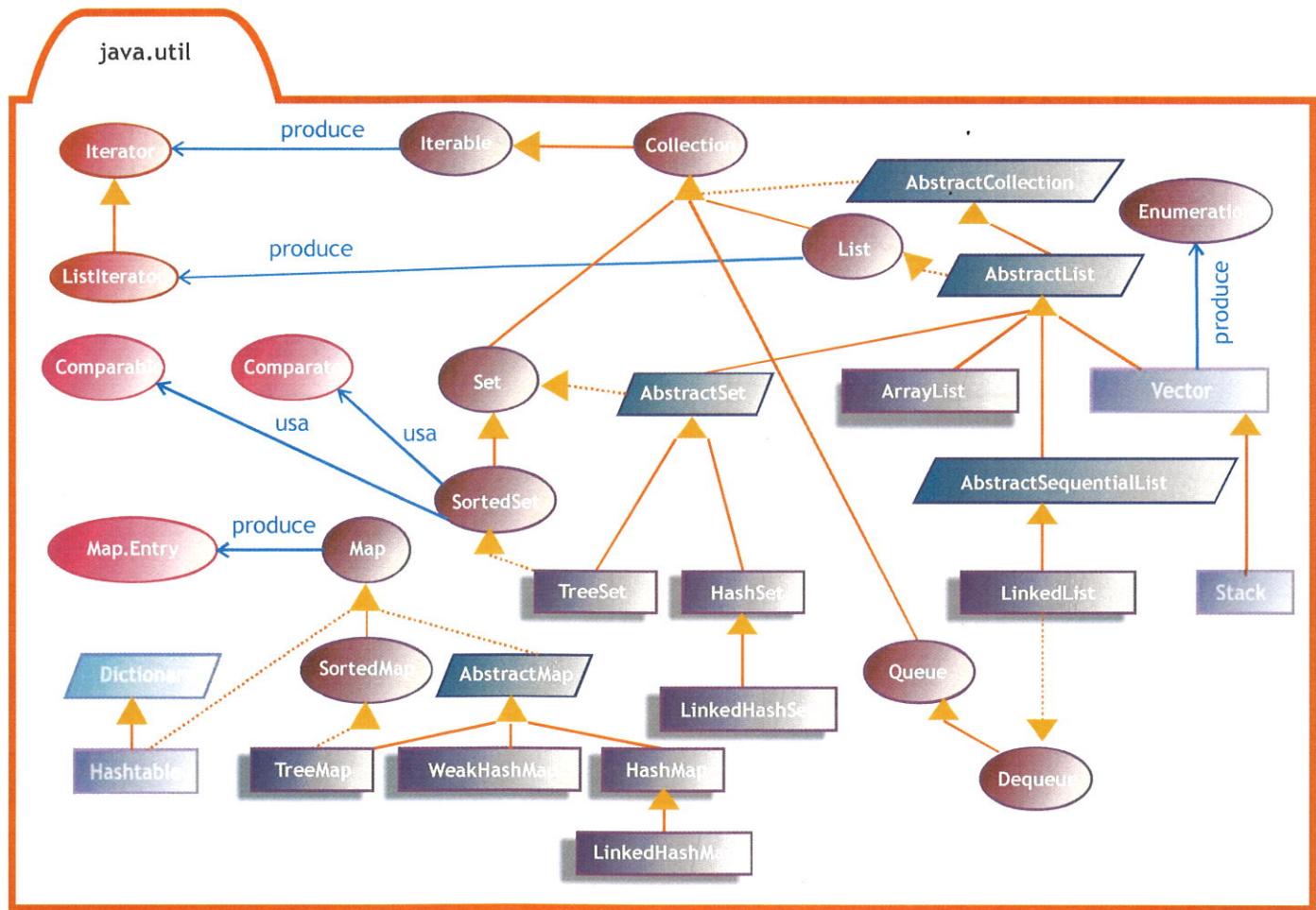


Ilustración 9-2, Diagrama de clases UML contenido los principales elementos de manejo de colecciones

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.

(9.1) 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 **dblemente 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





throw permite también *relanzar* excepciones. Esto significa que dentro de un catch podemos colocar una instrucción **throw** para lanzar la nueva excepción que será capturada por el catch correspondiente:

```
try{  
    ...  
} catch(ArrayIndexOutOfBoundsException exc){  
    throw new IOException();  
} catch(IOException){  
    ...  
}
```

El segundo catch capturará también las excepciones del primer tipo

(8.7) finally

La cláusula **finally** está pensada para limpiar el código en caso de excepción. Su uso es:

```
try{  
    ...  
} catch (FileNotFoundException fnfe){  
    ...  
} catch (IOException ioe){  
    ...  
} catch (Exception e){  
    ...  
} finally{  
    ...//Instrucciones de limpieza  
}
```

Las sentencias **finally** se ejecutan tras haberse ejecutado el catch correspondiente. Si ningún catch capturó la excepción, entonces se ejecutarán esas sentencias antes de devolver el control al siguiente nivel o antes de romperse la ejecución.

Hay que tener muy en cuenta que **las sentencias finally se ejecutan independientemente de si hubo o no excepción**. Es decir esas sentencias se ejecutan siempre, haya o no excepción. Son sentencias a ejecutarse en todo momento. Por ello se coloca en el bloque **finally** código común para todas las excepciones (y también para cuando no hay excepciones).

Ejercicio nº 41

Crear un programa que pida los datos de dos empleados.
De cada empleado nos interesa: nombre, edad y sueldo.
Se deben validar las entradas de datos volviendo a pedirlos.

Posteriormente se visualizaran los datos del empleado mas jove. Si los dos empleados tienen los mismos años, se visualizara la información de los dos.

Ejercicio nº 42 (pdf)

(8.4) métodos de la clase **Exception** Throwable

Como se observa en la Ilustración 8-1, la clase **Exception** es la superclase de todos los tipos de excepciones. Esto permite utilizar una serie de métodos comunes a todas las clases de excepciones:

* **String getMessage()**. Obtiene el mensaje descriptivo de la excepción o una indicación específica del error ocurrido:

```
try{  
}  
} catch (IOException ioe){  
    System.out.println(ioe.getMessage());  
}
```

* **String toString()**. Escribe una cadena sobre la situación de la excepción. Suele indicar la clase de excepción y el texto de **getMessage()**.

* **void printStackTrace()**. Escribe el método y mensaje de la excepción (la llamada información de pila). El resultado es el mismo mensaje que muestra el ejecutor (la máquina virtual de Java) cuando no se controla la excepción.

* **Throwable getCause()**. Causa del error, o null

(8.5) throws

Al llamar a métodos, ocurre un problema con las excepciones. El problema es, si el método da lugar a una excepción, ¿quién la maneja? ¿El propio método? O el código que hizo la llamada al método?

Con lo visto hasta ahora, sería el propio método quien se encargara de sus excepciones, pero esto complica el código ya que descentraliza el control de excepciones y dificulta el mantenimiento del código. Por eso otra posibilidad es hacer que la excepción la maneje el código que hizo la llamada.

Esto se hace añadiendo la palabra **throws** tras la primera línea de un método. Tras esa palabra se indica qué excepciones puede provocar el código del método. Si ocurre una excepción en el método, el código abandona ese método y regresa al código desde el que se llamó al método. Allí se posará en el catch apropiado para esa excepción. Ejemplo:

```
void usarArchivo (String archivo) throws IOException,  
InterruptedException {  
    ...  
}
```

En este caso se está indicando que el método **usarArchivo** puede provocar excepciones del tipo **IOException** y **InterruptedException**. Lo cual obliga a que el código que invoque a este método deba preparar el (o los) **catch**

Dentro del bloque **try** se colocan las instrucciones susceptibles de provocar una excepción, el bloque **catch** sirve para capturar esa excepción y evitar el fin de la ejecución del programa. Desde el bloque **catch** se maneja, en definitiva, la excepción.

OJO!!! → Cada **catch** maneja un tipo de excepción. Cuando se produce una excepción, se busca el **catch** que posea el manejador de excepción adecuado, será el que utilice el mismo tipo de excepción que se ha producido. Esto puede causar problemas si no se tiene cuidado, ya que la clase **Exception** es la superclase de todas las demás. Por lo que si se produjo, por ejemplo, una excepción de tipo **AritmeticException** y el primer **catch** captura el tipo genérico **Exception**, será ese **catch** el que se ejecute y no los demás.

Por eso el último **catch** debe ser el que capture excepciones genéricas y los primeros deben ser los más específicos. Lógicamente si vamos a tratar a todas las excepciones (sean del tipo que sean) igual, entonces basta con un solo **catch** que capture objetos **Exception**. Ejemplo (de mal uso):

```
int x;
try{
    x=Integer.parseInt(
        JOptionPane.showInputDialog("Escriba un número"));
}
catch (Exception e){
    JOptionPane.showMessageDialog(null, "Error indeterminado");
}
catch (NumberFormatException e){
    JOptionPane.showMessageDialog(null, "El número no es válido");
}
```

El código del recuadro no es alcanzable, porque siempre se ejecutaría el primer **catch**. De hecho hoy en día Java marca como error ese código. Lo correcto es:

```
int x;
try{
    x=Integer.parseInt(
        JOptionPane.showInputDialog("Escriba un número"));
}
catch (NumberFormatException e){
    JOptionPane.showMessageDialog(null, "El número no es válido");
}
catch (Exception e){
    JOptionPane.showMessageDialog(null, "Error indeterminado");
}
```

Ahora sí

- ◆ Se intenta dividir entre cero

Los errores de sintaxis son detectados durante la compilación, pero los errores de ejecución pueden provocar situaciones irreversibles, su control debe hacerse también en tiempo de ejecución y eso siempre ha sido problemático para la programación de aplicaciones.

En Java se puede preparar el código susceptible a provocar excepciones de modo que si ocurre una excepción, el código es *lanzado (throw)* a una determinada rutina previamente preparada por el programador, que permite manipular esa excepción. Si la excepción no fuera capturada, la ejecución del programa se detendría irremediablemente (en muchas ocasiones la propia sintaxis de Java impide que la excepción no sea controlada; es decir, obliga a controlarla).

En Java hay muchos tipos de excepciones (de operaciones de entrada y salida, de operaciones irreales,...). El paquete *java.lang.Exception* y sus subpaquetes contienen todos los tipos de excepciones.

Cuando se produce una excepción, se genera un objeto asociado a la misma. Este objeto es de clase *Exception* o de alguna de sus herederas. Este objeto se envía como parámetro al código que se ha definido para manejar la excepción. Dicho código puede manipular las propiedades del objeto *Exception*.

Hay una clase, la *java.lang.Error* y sus subclases que sirven para definir los errores irrecuperables más serios. Esos errores causan parada en el programa sin que el programador o programadora pueda evitarlo. Estos errores les produce el sistema y son incontrolables para el programador. Las excepciones son fallos más leves, y más manipulables.

(8.2) try y catch

El control de las excepciones se realiza mediante las sentencias **try** y **catch**. La sintaxis es:

```
try {  
    instrucciones que se ejecutan salvo que haya un error  
}  
catch (ClaseExcepción objetoQueCapturaLaExcepción) {  
    instrucciones que se ejecutan si hay un error  
}
```

Puede haber más de una sentencia **catch** para un mismo bloque **try**.



