2. Las ***clases internas locales*** tienen acceso a todas las variables miembro y métodos de la ***clase contenedora***. Pueden ver también los ***miembros heredados***, tanto por la ***clase interna local*** como por la ***clase contenedora***.
3. Las ***clases locales*** pueden utilizar las variables locales y argumentos de métodos ***visibles en ese bloque de código***, pero sólo si son ***final***[5] (en realidad la ***clase local*** trabaja con sus copias de las ***variables locales*** y por eso se exige que sean ***final*** y no puedan cambiar).
4. Un objeto de una ***clase interna local*** sólo puede existir en relación con un objeto de la ***clase contenedora***, que debe existir previamente.
5. La palabra ***this*** se puede utilizar en la misma forma que en las ***clases internas*** miembro, pero no las palabras ***new*** y ***super***.

Restricciones en el uso de las ***clases internas locales***:

1. No pueden tener el mismo nombre que ninguna de sus clases contenedoras.
2. No pueden definir variables, métodos y clases ***static***.
3. No pueden ser declaradas ***public***, ***protected***, ***private*** o ***package***, pues su visibilidad es siempre la de las variables locales, es decir, la del bloque en que han sido definidas.

Las ***clases internas locales*** se utilizan para definir clases ***Adapter*** en el AWT. A continuación se presenta un ejemplo de definición de clases internas locales:

```
// fichero ClasesIntLocales.java
// Este fichero demuestra cómo se crean clases locales

class A {
    int i=-1;    // variable miembro
    // constructor
    public A(int i) {this.i=i;}

    // definición de un método de la clase A
    public void getAi(final long k) {    // argumento final
        final double f=3.14;             // variable local final
        // definición de una clase interna local
        class BL {
            int j=2;
            public BL(int j) {this.j=j;} // constructor
            public void printBL() {
                System.out.println(" j="+j+" i="+i+" f="+f+" k="+k);
            }
        } // fin clase BL
        // se crea un objeto de BL
        BL bl = new BL(2*i);
        // se imprimen los datos de ese objeto
        bl.printBL();
    } // fin getAi
} // fin clase contenedora A


class ClasesIntLocales {
    public static void main(String [] arg) {
        // se crea dos objetos de la clase contenedora
        A a1 = new A(-10);
        A a2 = new A(-11);
```

[5] En Java 1.0 el cualificador ***final*** podía aplicarse a variables miembro, métodos y clases. En Java 1.1 puede también aplicarse a variables locales, argumentos de métodos e incluso al argumento de una ***exception***.

```
            // se llama al método getAi()
            a1.getAi(1000); // se crea y accede a un objeto de la clase local
            a2.getAi(2000);
        } // fin de main()

        public static void println(String str) {System.out.println(str);}

    }  // fin clase ClasesIntLocales
```

3.10.4 Clases anónimas

Las ***clases anónimas*** son muy similares a las ***clases internas locales***, pero ***sin nombre***. En las ***clases internas locales*** primero se define la clase y luego se crean uno o más objetos. En las ***clases anónimas*** se unen estos dos pasos: Como la clase no tiene nombre sólo se puede crear un único objeto, ya que las ***clases anónimas*** no pueden definir ***constructores***. Las clases anónimas se utilizan con mucha frecuencia en el AWT para definir clases y objetos que gestionen los eventos de los distintos componentes de la interface de usuario. No hay ***interfaces anónimas***.

Formas de definir una ***clase anónima***:

1. Las ***clases anónimas*** requieren una extensión de la palabra clave ***new***. Se definen en una expresión de ***Java***, incluida en una asignación o en la llamada a un método. Se incluye la palabra ***new*** seguida de la ***definición de la clase anónima***, entre llaves {…}.
2. Otra forma de definirlas es mediante la palabra ***new*** seguida del nombre de la clase de la que hereda (sin ***extends***) y la definición de la ***clase anónima*** entre llaves {…}. El nombre de la ***super-clase*** puede ir seguido de argumentos para su ***constructor*** (entre paréntesis, que con mucha frecuencia estarán vacíos pues se utilizará un constructor por defecto).
3. Una tercera forma de definirlas es con la palabra ***new*** seguida del nombre de la ***interface*** que implementa (sin ***implements***) y la definición de la ***clase anónima*** entre llaves {…}. En este caso la ***clase anónima*** deriva de ***Object***. El nombre de la ***interface*** va seguido por paréntesis vacíos, pues el constructor de ***Object*** no tiene argumentos.

Para las ***clases anónimas compiladas*** el compilador produce ficheros con un nombre del tipo ***ClaseContenedora$1.class***, asignando un número correlativo a cada una de las ***clases anónimas***.

Conviene ser muy cuidadoso respecto a los aspectos tipográficos de la definición de ***clases anónimas***, pues al no tener nombre dichas clases suelen resultar difíciles de leer e interpretar. Se aconseja utilizar las siguientes normas ***tipográficas***:

1. Se aconseja que la palabra ***new*** esté en la misma línea que el resto de la expresión.
2. Las ***llaves*** se abren en la misma línea que ***new***, después del cierre del paréntesis de los argumentos del constructor.
3. El ***cuerpo de la clase anónima*** se debe sangrar o indentar respecto a las líneas anteriores de código para que resulte claramente distinguible.
4. El ***cierre de las llaves*** va seguido por el ***resto de la expresión*** en la que se ha definido la ***clase anónima***. Esto puede servir como indicación tipográfica del cierre. Puede ser algo así como **};** o **});**

A continuación se presenta un ejemplo de definición de clase anónima en relación con el AWT:

```
unObjeto.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ...
    }
});
```

donde en negrita se señala la ***clase anónima***, que deriva de ***Object*** e implementa la interface ***ActionListener***.

Las ***clases anónimas*** se utilizan en lugar de ***clases locales*** para clases con muy poco código, de las que sólo hace falta un objeto. No pueden tener ***constructores***, pero sí ***inicializadores static*** o de ***objeto***. Además de las restricciones citadas, tienen ***restricciones*** similares a las ***clases locales***.

3.11 PERMISOS DE ACCESO EN JAVA

Una de las características de la *Programación Orientada a Objetos* es la ***encapsulación***, que consiste básicamente en ocultar la información que no es pertinente o necesaria para realizar una determinada tarea. Los permisos de acceso de ***Java*** son una de las herramientas para conseguir esta finalidad.

3.11.1 Accesibilidad de los packages

El primer tipo de accesibilidad hace referencia a la conexión física de los ordenadores y a los permisos de acceso entre ellos y en sus directorios y ficheros. En este sentido, un ***package*** es accesible si sus directorios y ficheros son accesibles (si están en un ordenador accesible y se tiene permiso de lectura). Además de la propia conexión física, serán accesibles aquellos ***packages*** que se encuentren en la variable ***CLASSPATH*** del sistema.

3.11.2 Accesibilidad de clases o interfaces

En principio, cualquier ***clase*** o ***interface*** de un ***package*** es accesible para todas las demás clases del ***package***, tanto si es ***public*** como si no lo es. Una clase ***public*** es accesible para cualquier otra clase siempre que su ***package*** sea accesible. Recuérdese que las ***clases*** e ***interfaces*** sólo pueden ser ***public*** o ***package*** (la opción por defecto cuando no se pone ningún modificador).

3.11.3 Accesibilidad de las variables y métodos miembros de una clase:

Desde dentro de la propia clase:

1. Todos los miembros de una clase son directamente accesibles (sin cualificar con ningún nombre o cualificando con la referencia ***this***) desde dentro de la propia clase. Los métodos no necesitan que las variables miembro sean pasadas como argumento.
2. Los miembros ***private*** de una clase sólo son accesibles para la propia clase.
3. Si el ***constructor*** de una clase es ***private***, sólo un método ***static*** de la propia clase puede crear objetos.

Desde una ***sub-clase***:

1. Las ***sub-clases*** heredan los miembros ***private*** de su ***super-clase***, pero sólo pueden acceder a ellos a través de métodos ***public***, ***protected*** o ***package*** de la super-clase.

Desde otras clases del ***package***:

1. Desde una clase de un ***package*** se tiene acceso a todos los miembros que no sean ***private*** de las demás clases del ***package***.

Desde otras clases fuera del ***package***:

1. Los métodos y variables son accesibles si la clase es ***public*** y el miembro es ***public***.
2. También son accesibles si la clase que accede es una ***sub-clase*** y el miembro es ***protected***.

La Tabla 3.1 muestra un resumen de los permisos de acceso en ***Java***.

Visibilidad	**public**	**protected**	**private**	**default**
Desde la propia clase	Sí	Sí	Sí	Sí
Desde otra clase en el propio package	Sí	Sí	No	Sí
Desde otra clase fuera del package	Sí	No	No	No
Desde una sub-clase en el propio package	Sí	Sí	No	Sí
Desde una sub-clase fuera del propio package	Sí	Sí	No	No

Tabla 3.1. Resumen de los permisos de acceso de Java.

3.12 TRANSFORMACIONES DE TIPO: CASTING

En muchas ocasiones hay que transformar una variable de un tipo a otro, por ejemplo de *int* a *double*, o de *float* a *long*. En otras ocasiones la conversión debe hacerse entre objetos de clases diferentes, aunque relacionadas mediante la herencia. En este apartado se explican brevemente estas transformaciones de tipo.

3.12.1 Conversión de tipos primitivos

La conversión entre tipos primitivos es más sencilla. En ***Java*** se realizan de modo automático conversiones implícitas ***de un tipo a otro de más precisión***, por ejemplo de ***int*** a ***long***, de ***float*** a ***double***, etc. Estas conversiones se hacen al mezclar variables de distintos tipos en expresiones matemáticas o al ejecutar sentencias de asignación en las que el miembro izquierdo tiene un tipo distinto (más amplio) que el resultado de evaluar el miembro derecho.

Las conversiones de un tipo de mayor a otro de menor precisión requieren una orden explícita del programador, pues son ***conversiones inseguras*** que pueden dar lugar a errores (por ejemplo, para pasar a ***short*** un número almacenado como ***int***, hay que estar seguro de que puede ser representado con el número de cifras binarias de ***short***). A estas conversiones explícitas de tipo se les llama ***cast***. El ***cast*** se hace poniendo el tipo al que se desea transformar entre paréntesis, como por ejemplo,

```
long result;
result = (long) (a/(b+c));
```

A diferencia de C/C++, en ***Java*** no se puede convertir un tipo numérico a ***boolean***.

La conversión de ***Strings*** (texto) a números se verá en el Apartado 4.3, en la página 64.

3.13 POLIMORFISMO

Ya se vio en el ejemplo presentado en el Apartado 1.3.8 y en los comentarios incluidos en qué consistía el ***polimorfismo***.

El ***polimorfismo*** tiene que ver con la relación que se establece entre la llamada a un método y el código que efectivamente se asocia con dicha llamada. A esta relación se llama ***vinculación*** (*binding*). La ***vinculación*** puede ser ***temprana*** (en tiempo de compilación) o ***tardía*** (en tiempo de ejecución). Con funciones normales o sobrecargadas se utiliza vinculación temprana (es posible y es lo más eficiente). Con funciones redefinidas en ***Java*** se utiliza siempre ***vinculación tardía***, excepto si el método es ***final***. El ***polimorfismo*** es la ***opción por defecto*** en ***Java***.

La ***vinculación tardía*** hace posible que, con un método declarado en una clase base (o en una interface) y redefinido en las clases derivadas (o en clases que implementan esa interface), sea ***el tipo de objeto*** y ***no el tipo de la referencia*** lo que determine qué definición del método se va a utilizar. El tipo del objeto al que apunta una referencia sólo puede conocerse en tiempo de ejecución, y por eso el ***polimorfismo*** necesita evaluación tardía.

El ***polimorfismo*** permite a los programadores separar las cosas que cambian de las que no cambian, y de esta manera hacer más fácil la ampliación, el mantenimiento y la reutilización de los programas.

El ***polimorfismo*** puede hacerse con referencias de ***super-clases abstract***, ***super-clases normales*** e ***interfaces***. Por su mayor flexibilidad y por su independencia de la jerarquía de clases estándar, las ***interfaces*** permiten ampliar muchísimo las posibilidades del polimorfismo.

3.13.1 Conversión de objetos

El ***polimorfismo*** visto previamente está basado en utilizar referencias de un tipo más “amplio” que los objetos a los que apuntan. Las ventajas del polimorfismo son evidentes, pero hay una importante limitación: el tipo de la referencia (clase abstracta, clase base o interface) limita los métodos que se pueden utilizar y las variables miembro a las que se pueden acceder. Por ejemplo, un objeto puede tener una referencia cuyo tipo sea una ***interface***, aunque sólo en el caso en que su ***clase*** o ***una de sus super-clases*** implemente dicha ***interface***. Un objeto cuya referencia es un tipo ***interface*** sólo puede utilizar los métodos definidos en dicha ***interface***. Dicho de otro modo, ese objeto no puede utilizar las variables y los métodos propios de su clase. De esta forma las ***referencias de tipo interface*** definen, limitan y unifican la forma de utilizarse de objetos pertenecientes a clases muy distintas (que implementan dicha interface).

Si se desea utilizar todos los métodos y acceder a todas las variables que la clase de un objeto permite, hay que utilizar un ***cast explícito***, que convierta su referencia más general en la del tipo específico del objeto. De aquí una parte importante del interés del ***cast*** entre objetos (más bien entre referencias, habría que decir).

Para la conversión entre objetos de distintas clases, ***Java*** exige que dichas clases estén relacionadas por ***herencia*** (una deberá ser ***sub-clase*** de la otra). Se realiza una ***conversión implícita*** o ***automática*** de una ***sub-clase*** a una ***super-clase*** siempre que se necesite, ya que el objeto de la ***sub-clase*** siempre tiene toda la información necesaria para ser utilizado en lugar de un objeto de la ***super-clase***. No importa que la ***super-clase*** no sea capaz de contener toda la información de la ***sub-clase***.

La conversión en sentido contrario -utilizar un objeto de una ***super-clase*** donde se espera encontrar uno de la ***sub-clase***- debe hacerse de modo ***explícito*** y puede producir errores por falta de información o de métodos. Si falta información, se obtiene una ***ClassCastException***.

No se puede acceder a las variables exclusivas de la sub-clase a través de una referencia de la super-clase. Sólo se pueden utilizar los métodos definidos en la super-clase, aunque la definición utilizada para dichos métodos sea la de la sub-clase.

Por ejemplo, supóngase que se crea un objeto de una ***sub-clase B*** y se referencia con un nombre de una ***super-clase A***,

```
A a = new B();
```

en este caso el objeto creado dispone de más información de la que la referencia ***a*** le permite acceder (podría ser, por ejemplo, una nueva variable miembro ***j*** declarada en ***B***). Para acceder a esta información adicional hay que hacer un ***cast*** explícito en la forma ***(B)a***. Para imprimir esa variable ***j*** habría que escribir (los paréntesis son necesarios):

```
System.out.println( ((B)a).j );
```

Un ***cast*** de un objeto a la ***super-clase*** puede permitir utilizar variables -no métodos- de la ***super-clase***, aunque estén redefinidos en la ***sub-clase***. Considérese el siguiente ejemplo: La clase ***C*** deriva de ***B*** y ***B*** deriva de ***A***. Las tres definen una variable ***x***. En este caso, si desde el código de la sub-clase ***C*** se utiliza:

```
x               // se accede a la x de C
this.x          // se accede a la x de C
super.x         // se accede a la x de B. Sólo se puede subir un nivel
((B)this).x     // se accede a la x de B
((A)this).x     // se accede a la x de A
```

4. CLASES DE UTILIDAD

Programando en ***Java*** nunca se parte de cero: siempre se parte de la infraestructura definida por el API de ***Java***, cuyos ***packages*** proporcionan una buena base para que el programador construya sus aplicaciones. En este Capítulo se describen algunas clases que serán de utilidad para muchos programadores.

4.1 ARRAYS

Los ***arrays*** de ***Java*** (vectores, matrices, hiper-matrices de más de dos dimensiones) se tratan como objetos de una clase predefinida. Los ***arrays*** son ***objetos***, pero con algunas características propias. Los ***arrays*** pueden ser asignados a objetos de la clase ***Object*** y los métodos de ***Object*** pueden ser utilizados con ***arrays***.

Algunas de sus características más importantes de los ***arrays*** son las siguientes:

1. Los ***arrays*** se crean con el operador ***new*** seguido del tipo y número de elementos.
2. Se puede acceder al número de elementos de un array con la variable miembro implícita ***length*** (por ejemplo, ***vect.length***).
3. Se accede a los elementos de un ***array*** con los ***corchetes []*** y un ***índice*** que varía de 0 a ***length-1***.
4. Se pueden crear ***arrays*** de objetos de cualquier tipo. En principio un ***array*** de objetos es un ***array de referencias*** que hay que completar llamando al operador ***new***.
5. Los elementos de un ***array*** se inicializan al valor por defecto del tipo correspondiente (cero para valores numéricos, el carácter nulo para ***char***, ***false*** para ***boolean***, ***null*** para ***Strings*** y para referencias).
6. Como todos los objetos, los ***arrays*** se pasan como argumentos a los métodos ***por referencia***.
7. Se pueden crear ***arrays anónimos*** (por ejemplo, crear un nuevo array como argumento actual en la llamada a un método).

Inicialización de ***arrays***:

1. Los ***arrays*** se pueden inicializar con valores entre llaves {...} separados por comas.
2. También los ***arrays de objetos*** se pueden inicializar con varias llamadas a ***new*** dentro de unas llaves {...}.
3. Si se igualan dos referencias a un array no se copia el array, sino que se tiene un array con dos nombres, apuntando al mismo y único objeto.
4. Creación de una ***referencia*** a un array. Son posibles dos formas:

```
double[] x; // preferible
double x[];
```

5. Creación del ***array*** con el operador ***new***:

```
x = new double[100];
```

6. Las dos etapas 4 y 5 se pueden unir en una sola:

```
double[] x = new double[100];
```

A continuación se presentan algunos ejemplos de creación de arrays:

```
// crear un array de 10 enteros, que por defecto se inicializan a cero
int v[] = new int[10];
// crear arrays inicializando con determinados valores
int v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
String dias[] = {"lunes", "martes", "miercoles", "jueves",
                 "viernes", "sabado", "domingo"};
// array de 5 objetos
MiClase listaObj[] = new MiClase[5]; // de momento hay 5 referencias a null
for( i = 0 ; i < 5;i++)
    listaObj[i] = new MiClase(...);
// array anónimo
obj.metodo(new String[]={"uno", "dos", "tres"});
```

4.1.1 Arrays bidimensionales

Los arrays bidimensionales de ***Java*** se crean de un modo muy similar al de C++ (con reserva dinámica de memoria). En ***Java*** una ***matriz*** es un ***vector*** de ***vectores fila***, o más en concreto un vector de referencias a los vectores fila. Con este esquema, cada fila podría tener un número de elementos diferente.

Una matriz se puede crear directamente en la forma,

```
int [][] mat = new int[3][4];
```

o bien se puede crear de modo dinámico dando los siguientes pasos:

1. Crear la ***referencia*** indicando con un doble corchete que es una ***referencia a matriz***,

   ```
   int[][] mat;
   ```

2. Crear el vector de referencias a las filas,

   ```
   mat = new int[nfilas][];
   ```

3. Reservar memoria para los vectores correspondientes a las filas,

   ```
   for (int i=0; i<nfilas; i++);
       mat[i] = new int[ncols];
   ```

A continuación se presentan algunos ejemplos de creación de arrays bidimensionales:

```
// crear una matriz 3x3
//     se inicializan a cero
double mat[][] = new double[3][3];
int [][] b = {{1, 2, 3},
              {4, 5, 6}, // esta coma es permitida
              };
int c = new[3][]; // se crea el array de referencias a arrays
c[0] = new int[5];
c[1] = new int[4];
c[2] = new int[8];
```

En el caso de una matriz ***b***, ***b.length*** es el número de filas y ***b[0].length*** es el número de columnas (de la fila 0). Por supuesto, los arrays bidimensionales pueden contener tipos primitivos de cualquier tipo u objetos de cualquier clase.

4.2 CLASES STRING Y STRINGBUFFER

Las clases ***String*** y ***StringBuffer*** están orientadas a manejar cadenas de caracteres. La clase ***String*** está orientada a manejar cadenas de caracteres constantes, es decir, que no pueden cambiar. La clase ***StringBuffer*** permite que el programador cambie la cadena insertando, borrando, etc. La primera es más eficiente, mientras que la segunda permite más posibilidades.

Ambas clases pertenecen al package ***java.lang***, y por lo tanto no hay que importarlas. Hay que indicar que el ***operador de concatenación*** (+) entre objetos de tipo ***String*** utiliza internamente objetos de la clase ***StringBuffer*** y el método ***append()***.

Los métodos de ***String*** se pueden utilizar directamente sobre ***literals*** (cadenas entre comillas), como por ejemplo: *"Hola".length()*.

4.2.1 Métodos de la clase String

Los objetos de la clase ***String*** se pueden crear a partir de cadenas constantes o *literals*, definidas entre dobles comillas, como por ejemplo: "Hola". ***Java*** crea siempre un objeto ***String*** al encontrar una cadena entre comillas. A continuación se describen dos formas de crear objetos de la clase ***String***,

```
String str1 = "Hola";             // el sistema más eficaz de crear Strings
String str2 = new String("Hola"); // también se pueden crear con un constructor
```

El primero de los métodos expuestos es el más eficiente, porque como al encontrar un texto entre comillas se crea automáticamente un objeto ***String***, en la práctica utilizando ***new*** se llama al constructor dos veces. También se pueden crear objetos de la clase ***String*** llamando a otros constructores de la clase, a partir de objetos ***StringBuffer***, y de arrays de ***bytes*** o de ***chars***.

La Tabla 4.1 muestra los métodos más importantes de la clase ***String***.

Métodos de String	**Función que realizan**
String(...)	Constructores para crear Strings a partir de arrays de bytes o de caracteres (ver documentación on-line)
String(String str) y String(StringBuffer sb)	Costructores a partir de un objeto String o StringBuffer
charAt(int)	Devuelve el carácter en la posición especificada
getChars(int, int, char[], int)	Copia los caracteres indicados en la posición indicada de un array de caracteres
indexOf(String, [int])	Devuelve la posición en la que aparece por primera vez un String en otro String, a partir de una posición dada (opcional)
lastIndexOf(String, [int])	Devuelve la última vez que un String aparece en otro empezando en una posición y hacia el principio
length()	Devuelve el número de caracteres de la cadena
replace(char, char)	Sustituye un carácter por otro en un String
startsWith(String)	Indica si un String comienza con otro String o no
substring(int, int)	Devuelve un String extraído de otro
toLowerCase()	Convierte en minúsculas (puede tener en cuenta el locale)
toUpperCase()	Convierte en mayúsculas (puede tener en cuenta el locale)
trim()	Elimina los espacios en blanco al comienzo y final de la cadena
valueOf()	Devuelve la representación como String de sus argumento. Admite Object, arrays de caracteres y los tipos primitivos

Tabla 4.1. Algunos métodos de la clase String.

Un punto importante a tener en cuenta es que hay métodos, tales como ***System.out.println()***, que exigen que su argumento sea un objeto de la clase ***String***. Si no lo es, habrá que utilizar algún metodo que lo convierta en ***String***.

El ***locale***, citado en la Tabla 4.1, es la forma que java tiene para adaptarse a las peculiaridades de los idiomas distintos del inglés: acentos, caracteres especiales, forma de escribir las fechas y las horas, unidades monetarias, etc.

4.2.2 Métodos de la clase StringBuffer

La clase ***StringBuffer*** se utiliza prácticamente siempre que se desee modificar una cadena de caracteres. Completa los métodos de la clase ***String*** ya que éstos realizan sólo operaciones sobre el texto que no conllevan un aumento o disminución del número de letras del ***String***.

Recuérdese que hay muchos métodos cuyos argumentos deben ser objetos ***String***, que antes de pasar esos argumentos habrá que realizar la conversión correspondiente. La Tabla 4.2 muestra los métodos más importantes de la clase ***StringBuffer***.

Métodos de StringBuffer	**Función que realizan**
StringBuffer(), StringBuffer(int), StringBuffer(String)	Constructores
append(...)	Tiene muchas definiciones diferentes para añadir un String o una variable (int, long, double, etc.) a su objeto
capacity()	Devuelve el espacio libre del StringBuffer
charAt(int)	Devuelve el carácter en la posición especificada
getChars(int, int, char[], int)	Copia los caracteres indicados en la posición indicada de un array de caracteres
insert(int,)	Inserta un String o un valor (int, long, double, ...) en la posición especificada de un StringBuffer
length()	Devuelve el número de caracteres de la cadena
reverse()	Cambia el orden de los caracteres
setCharAt(int, char)	Cambia el carácter en la posición indicada
setLength(int)	Cambia el tamaño de un StringBuffer
toString()	Convierte en objeto de tipo String

Tabla 4.2. Algunos métodos de la clase StringBuffer.

4.3 WRAPPERS

Los ***Wrappers*** (*envoltorios*) son clases diseñadas para ser un ***complemento*** de los ***tipos primitivos***. En efecto, los tipos primitivos son los únicos elementos de ***Java*** que no son objetos. Esto tiene algunas ventajas desde el punto de vista de la ***eficiencia***, pero algunos inconvenientes desde el punto de vista de la ***funcionalidad***. Por ejemplo, los ***tipos primitivos*** siempre se pasan como argumento a los métodos ***por valor***, mientras que los ***objetos*** se pasan ***por referencia***. No hay forma de modificar en un método un argumento de tipo primitivo y que esa modificación se trasmita al entorno que hizo la llamada. Una forma de conseguir esto es utilizar un ***Wrapper***, esto es un objeto cuya variable miembro es el tipo primitivo que se quiere modificar. Las clases ***Wrapper*** también proporcionan métodos para realizar otras tareas con lo tipos primitivos, tales como conversión con cadenas de caracteres en uno y otro sentido.

Existe una clase ***Wrapper*** para cada uno de los tipos primitivos numéricos, esto es, existen las clases ***Byte***, ***Short***, ***Integer***, ***Long***, ***Float*** y ***Double*** (obsérvese que los nombres empiezan por mayúscula, siguiendo la nomenclatura típica de ***Java***). A continuación se van a ver dos de estas clases: ***Double*** e ***Integer***. Las otras cuatro son similares y sus características pueden consultarse en la documentación on-line.

4.3.1 Clase Double

La clase ***java.lang.Double*** deriva de ***Number***, que a su vez deriva de ***Object***. Esta clase contiene un valor primitivo de tipo ***double***. La Tabla 4.3 muestra algunos métodos y constantes predefinidas de la clase ***Double***.

Métodos	**Función que desempeñan**
Double(double) y Double(String)	Los constructores de esta clase
doubleValue(), floatValue(), longValue(), intValue(), shortValue(), byteValue()	Métodos para obtener el valor del tipo primitivo
String toString(), Double valueOf(String)	Conversores con la clase String
isInfinite(), isNaN()	Métodos de chequear condiiciones
equals(Object)	Compara con otro objeto
MAX_DOUBLE, MIN_DOUBLE, POSITIVE_INFINITY, NEGATIVE_INFINITY, NaN, TYPE	Constantes predefinidas. TYPE es el objeto Class representando esta clase

Tabla 4.3. Métodos y constantes de la clase Double.

El *Wrapper* ***Float*** es similar al *Wrapper* ***Double***.

4.3.2 Clase Integer

La clase ***java.lang.Integer*** tiene como variable miembro un valor de tipo ***int***. La Tabla 4.4 muestra los métodos y constantes de la clase ***Integer***.

Métodos	**Función que desempeñan**
Integer(int) y Integer(String)	Constructores de la clase
doubleValue(), floatValue(), longValue(), intValue(), shortValue(), byteValue()	Conversores con otros tipos primitivos
Integer decode(String), Integer parseInt(String), String toString(), Integer valueOf(String)	Conversores con String
String toBinaryString(int), String toHexString(int), String toOctalString(int)	Conversores a cadenas representando enteros en otros sistemas de numeración
Integer getInteger(String)	Determina el valor de una propiedad del sistema a partir del nombre de dicha propiedad
MAX_VALUE, MIN_VALUE, TYPE	Constantes predefinidas

Tabla 4.4. Métodos y constantes de la clase Integer.

Los *Wrappers* ***Byte***, ***Short*** y ***Long*** son similares a ***Integer***.

Los *Wrappers* son utilizados para convertir cadenas de caracteres (texto) en números. Esto es útil cuando se leen valores desde el teclado, desde un fichero de texto, etc. Los ejemplos siguientes muestran algunas conversiones:

```
String numDecimalString = "8.978";
float  numFloat=Float.valueOf(numDecimalString).floatValue();   // numFloat = 8,979
double numDouble=Double.valueOf(numDecimalString).doubleValue();// numDouble = 8,979
String numIntString = "1001";
int    numInt=Integer.valueOf(numIntString).intValue();         // numInt = 1001
```

En el caso de que el texto no se pueda convertir directamente al tipo especificado se lanza una excepción de tipo ***NumberFormatException***, por ejemplo si se intenta convertir directamente el texto “4.897” a un número entero. El proceso que habrá que seguir será convertirlo en primer lugar a un número ***float*** y posteriormente a número entero.

4.4 CLASE MATH

La clase ***java.lang.Math*** deriva de ***Object***. La clase ***Math*** proporciona métodos ***static*** para realizar las operaciones matemáticas más habituales. Proporciona además las constantes ***E*** y ***PI***, cuyo significado no requiere muchas explicaciones.

La Tabla 4.5 muestra los métodos matemáticos soportados por esta clase.

Métodos	**Significado**	**Métodos**	**Significado**
abs()	Valor absoluto	sin(double)	Calcula el seno
acos()	Arcocoseno	tan(double)	Calcula la tangente
asin()	Arcoseno	exp()	Calcula la función exponencial
atan()	Arcotangente entre -PI/2 y PI/2	log()	Calcula el logaritmo natural (base e)
atan2(,)	Arcotangente entre -PI y PI	max(,)	Máximo de dos argumentos
ceil()	Entero más cercano en dirección a infinito	min(,)	Mínimo de dos argumentos
floor()	Entero más cercano en dirección a -infinito	random()	Número aleatorio entre 0.0 y 1.0
round()	Entero más cercano al argumento	power(,)	Devuelve el primer argumento elevado al segundo
rint(double)	Devuelve el entero más próximo	sqrt()	Devuelve la raíz cuadrada
IEEEremainder(double , double)	Calcula el resto de la división	toDegrees(double)	Pasa de radianes a grados (Java 2)
cos(double)	Calcula el coseno	toRadians()	Pasa de grados a radianes (Java 2)

Tabla 4.5. Métodos matemáticos de la clase Math.

4.5 COLECCIONES

Java dispone también de clases e interfaces para trabajar con colecciones de objetos. En primer lugar se verán las clases ***Vector*** y ***Hashtable***, así como la interface ***Enumeration***. Estas clases están presentes en lenguaje desde la primera versión. Después se explicará brevemente la ***Java Collections Framework***, introducida en la versión JDK 1.2.

4.5.1 Clase Vector

La clase ***java.util.Vector*** deriva de ***Object***, implementa ***Cloneable*** (para poder sacar copias con el método ***clone()***) y ***Serializable*** (para poder ser convertida en cadena de caracteres).

Como su mismo nombre sugiere, ***Vector*** representa un ***array de objetos*** (referencias a objetos de tipo ***Object***) que puede crecer y reducirse, según el número de elementos. Además permite acceder a los elementos con un ***índice***, aunque no permite utilizar los corchetes [].

El método ***capacity()*** devuelve el tamaño o número de elementos que puede tener el vector. El método ***size()*** devuelve el número de elementos que realmente contiene, mientras que ***capacityIncrement*** es una variable que indica el salto que se dará en el tamaño cuando se necesite crecer. La Tabla 4.6 muestra los métodos más importantes de la clase ***Vector***. Puede verse que el gran número de métodos que existen proporciona una notable flexibilidad en la utilización de esta clase.

Además de ***capacityIncrement***, existen otras dos variables miembro: ***elementCount***, que representa el número de componentes válidos del vector, y ***elementData[]*** que es el array de ***Objects*** donde realmente se guardan los elementos del objeto ***Vector*** (***capacity*** es el tamaño de este array). Las tres variables citadas son ***protected***.

Métodos	**Función que realizan**
Vector(), Vector(int), Vector(int, int)	Constructores que crean un vector vacío, un vector de la capacidad indicada y un vector de la capacidad e incremento indicados
void addElement(Object obj)	Añade un objeto al final
boolean removeElement(Object obj)	Elimina el primer objeto que encuentra como su argumento y desplaza los restantes. Si no lo encuentra devuelve false
void removeAllElements()	Elimina todos los elementos
Object clone()	Devuelve una copia del vector
void copyInto(Object anArray[])	Copia un vector en un array
void trimToSize()	Ajusta el tamaño a los elementos que tiene
void setSize(int newSize)	Establece un nuevo tamaño
int capacity()	Devuelve el tamaño (capacidad) del vector
int size()	Devuelve el número de elementos
boolean isEmpty()	Devuelve true si no tiene elementos
Enumeration elements()	Devuelve una Enumeración con los elementos
boolean contains(Object elem)	Indica si contiene o no un objeto
int indexOf(Object elem, int index)	Devuelve la posición de la primera vez que aparece un objeto a partir de una posición dada
int lastIndexOf(Object elem, int index)	Devuelve la posición de la última vez que aparece un objeto a partir de una posición, hacia atrás
Object elementAt(int index)	Devuelve el objeto en una determinada posición
Object firstElement()	Devuelve el primer elemento
Object lastElement()	Devuelve el último elemento
void setElementAt(Object obj, int index)	Cambia el elemento que está en una determinada posición
void removeElementAt(int index)	Elimina el elemento que está en una determinada posición
void insertElementAt(Object obj, int index)	Inserta un elemento por delante de una determinada posición

Tabla 4.6. Métodos de la clase Vector.

4.5.2 Interface Enumeration

La interface ***java.util.Enumeration*** define métodos útiles para recorrer una colección de objetos. Puede haber distintas clases que implementen esta interface y todas tendrán un comportamiento similar.

La interface ***Enumeration*** declara dos métodos:

1. ***public boolean hasMoreElements()***. Indica si hay más elementos en la colección o si se ha llegado ya al final.
2. ***public Object nextElement()***. Devuelve el siguiente objeto de la colección. Lanza una *NoSuchElementException* si se llama y ya no hay más elementos.

Ejemplo: Para imprimir los elementos de un vector ***vec*** se pueden utilizar las siguientes sentencias:

```
for (Enumeration e = vec.elements(); e.hasMoreElements(); ) {
    System.out.println(e.nextElement());
}
```

donde, como puede verse en la Tabla 4.6, el método ***elements()*** devuelve precisamente una referencia de tipo ***Enumeration***. Con los métodos ***hasMoreElements()*** y ***nextElement()*** y un bucle ***for*** se pueden ir imprimiendo los distintos elementos del objeto ***Vector***.

4.5.3 Clase Hashtable

La clase ***java.util.Hashtable*** extiende ***Dictionary*** (***abstract***) e implementa ***Cloneable*** y ***Serializable***. Una ***hash table*** es una tabla que relaciona una ***clave*** con un ***valor***. Cualquier objeto distinto de ***null*** puede ser tanto ***clave*** como ***valor***.

La clase a la que pertenecen las ***claves*** debe implementar los métodos ***hashCode()*** y ***equals()***, con objeto de hacer búsquedas y comparaciones. El método ***hashCode()*** devuelve un entero único y distinto para cada clave, que es siempre el mismo en una ejecución del programa pero que puede cambiar de una ejecución a otra. Además, para dos claves que resultan iguales según el método ***equals()***, el método ***hashCode()*** devuelve el mismo entero. Las ***hash tables*** están diseñadas para mantener una colección de pares ***clave/valor***, permitiendo insertar y realizar búsquedas de un modo muy eficiente

Cada objeto de ***Hashtable*** tiene dos variables: ***capacity*** y ***load factor*** (entre 0.0 y 1.0). Cuando el número de elementos excede el producto de estas variables, la ***Hashtable*** crece llamando al método ***rehash()***. Un ***load factor*** más grande apura más la memoria, pero será menos eficiente en las búsquedas. Es conveniente partir de una ***Hashtable*** suficientemente grande para no estar ampliando continuamente.

Ejemplo de definición de ***Hashtable***:

```
Hashtable numeros = new Hashtable();
numbers.put("uno", new Integer(1));
numbers.put("dos", new Integer(2));
numbers.put("tres", new Integer(3));
```

donde se ha hecho uso del método ***put()***. La Tabla 4.7 muestra los métodos de la clase ***Hashtable***.

Métodos	**Función que realizan**
Hashtable(), Hashtable(int nElements), Hashtable(int nElements, float loadFactor)	Constructores
int size()	Devuelve el tamaño de la tabla
boolean isEmpty()	Indica si la tabla está vacía
Enumeration keys()	Devuelve una Enumeration con las claves
Enumeration elements()	Devuelve una Enumeration con los valores
boolean contains(Object value)	Indica si hay alguna clave que se corresponde con el valor
boolean containsKey(Object key)	Indica si existe esa clave en la tabla
Object get(Object key)	Devuelve un valor dada la clave
void rehash()	Amplía la capacidad de la tabla
Object put(Object key, Object value)	Establece una relación clave-valor
Object remove(Object key)	Elimina un valor por la clave
void clear()	Limpia la tabla
Object clone()	Hace una copia de la tabla
String toString()	Devuelve un string representando la tabla

Tabla 4.7. Métodos de la clase Hashtable.

4.5.4 El Collections Framework de Java 1.2

En la versión 1.2 del JDK se introdujo el ***Java Framework Collections*** o “estructura de colecciones de Java” (en adelante JCF). Se trata de un conjunto de clases e interfaces que mejoran notablemente las capacidades del lenguaje respecto a estructuras de datos. Además, constituyen un excelente ejemplo de aplicación de los conceptos propios de la *programación orientada a objetos*. Dada la amplitud de ***Java*** en éste y en otros aspectos se va a optar por insistir en la descripción general,

dejando al lector la tarea de buscar las características concretas de los distintos métodos en la documentación de ***Java***. En este apartado se va a utilizar una forma -más breve que las tablas utilizadas en otros apartados- de informar sobre los métodos disponibles en una clase o interface.

La Figura 4.1 muestra la jerarquía de interfaces de la ***Java Collection Framework*** (JCF). En letra cursiva se indican las clases que implementan las correspondientes interfaces. Por ejemplo, hay dos clases que implementan la interface ***Map***: ***HashMap*** y ***Hashtable***.

Las clases vistas en los apartados anteriores son clases "históricas", es decir, clases que existían antes de la versión JDK 1.2. Dichas clases se denotan en la Figura 4.1 con la letra "h" entre paréntesis. Aunque dichas clases se han mantenido por motivos de compatibilidad, sus métodos no siguen las reglas del diseño general del JCF; en la medida de lo posible se recomienda utilizar las nuevas clases.

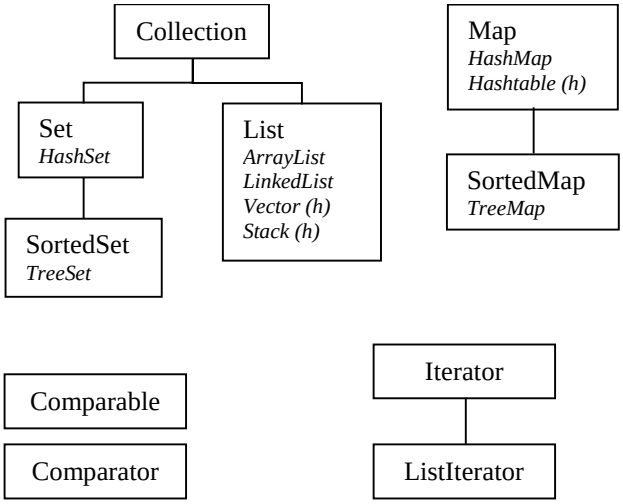

Figura 4.1. Interfaces de la Collection Framework.

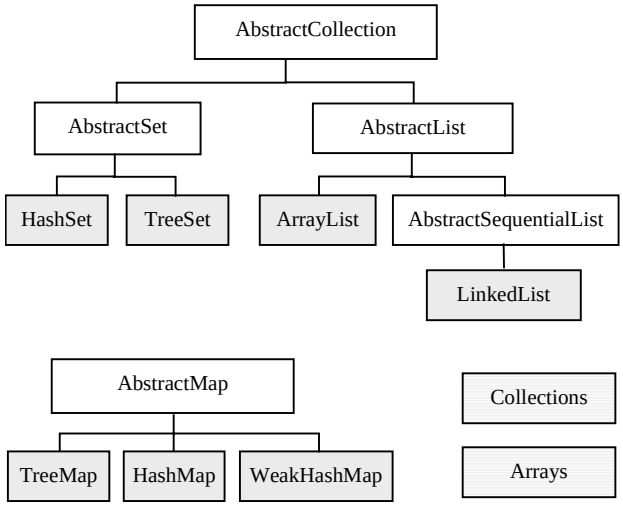

Figura 4.2. Jerarquía de clases de la Collection Framework.

En el diseño de la JCF las ***interfaces*** son muy importantes porque son ellas las que determinan las capacidades de las clases que las implementan. Dos clases que implementan la misma interface se pueden utilizar exactamente de la misma forma. Por ejemplo, las clases ***ArrayList*** y ***LinkedList*** disponen exactamente de los mismos métodos y se pueden utilizar de la misma forma. La diferencia está en la implementación: mientras que ***ArrayList*** almacena los objetos en un array, la clase ***LinkedList*** los almacena en una lista vinculada. La primera será más eficiente para acceder a un elemento arbitrario, mientras que la segunda será más flexible si se desea borrar e insertar elementos.

La Figura 4.2 muestra la jerarquía de clases de la JCF. En este caso, la jerarquía de clases es menos importante desde el punto de vista del usuario que la jerarquía de interfaces. En dicha figura se muestran con fondo blanco las clases abstractas, y con fondo gris claro las clases de las que se pueden crear objetos.

Las clases ***Collections*** y ***Arrays*** son un poco especiales: no son ***abstract***, pero no tienen constructores públicos con los que se puedan crear objetos. Fundamentalmente contienen métodos ***static*** para realizar ciertas operaciones de utilidad: ordenar, buscar, introducir ciertas características en objetos de otras clases, etc.

4.5.4.1 Elementos del Java Collections Framework

Interfaces de la JCF: Constituyen el elemento central de la JCF.

- ***Collection***: define métodos para tratar una colección genérica de elementos
- ***Set***: colección que no admite elementos repetidos
- ***SortedSet***: *set* cuyos elementos se mantienen ordenados según el criterio establecido
- ***List***: admite elementos repetidos y mantiene un orden inicial
- ***Map***: conjunto de pares clave/valor, sin repetición de claves
- ***SortedMap***: *map* cuyos elementos se mantienen ordenados según el criterio establecido

Interfaces de soporte:

- ***Iterator***: sustituye a la interface ***Enumeration***. Dispone de métodos para recorrer una colección y para borrar elementos.
- ***ListIterator***: deriva de ***Iterator*** y permite recorrer *lists* en ambos sentidos.
- ***Comparable***: declara el método ***compareTo()*** que permite ordenar las distintas colecciones según un orden natural (***String***, ***Date***, ***Integer***, ***Double***, …).
- ***Comparator***: declara el método ***compare()*** y se utiliza en lugar de ***Comparable*** cuando se desea ordenar objetos no estándar o sustituir a dicha interface.

Clases de propósito general: Son las implementaciones de las interfaces de la JFC.

- ***HashSet***: Interface ***Set*** implementada mediante una hash table.
- ***TreeSet***: Interface ***SortedSet*** implementada mediante un árbol binario ordenado.
- ***ArrayList***: Interface ***List*** implementada mediante un array.
- ***LinkedList***: Interface ***List*** implementada mediante una lista vinculada.
- ***HashMap***: Interface ***Map*** implementada mediante una hash table.
- ***WeakHashMap***: Interface ***Map*** implementada de modo que la memoria de los pares clave/valor pueda ser liberada cuando las claves no tengan referencia desde el exterior de la ***WeakHashMap***.
- ***TreeMap***: Interface ***SortedMap*** implementada mediante un árbol binario

Clases Wrapper: Colecciones con características adicionales, como no poder ser modificadas o estar sincronizadas. No se accede a ellas mediante constructores, sino mediante métodos “factory” de la clase ***Collections***.

Clases de utilidad: Son mini-implementaciones que permiten obtener ***sets*** especializados, como por ejemplo ***sets*** constantes de un sólo elemento (***singleton***) o ***lists*** con ***n*** copias del mismo elemento (***nCopies***). Definen las constantes EMPTY_SET y EMPTY_LIST. Se accede a través de la clase ***Collections***.

Clases históricas: Son las clases ***Vector*** y ***Hashtable*** presentes desde las primeras versiones de ***Java***. En las versiones actuales, implementan respectivamente las interfaces ***List*** y ***Map***, aunque conservan también los métodos anteriores.

Clases abstractas: Son las clases abstract de la Figura 4.2. Tienen total o parcialmente implementados los métodos de la interface correspondiente. Sirven para que los usuarios deriven de ellas sus propias clases con un mínimo de esfuerzo de programación.

Algoritmos: La clase ***Collections*** dispone de métodos ***static*** para ordenar, desordenar, invertir orden, realizar búsquedas, llenar, copiar, hallar el mínimo y hallar el máximo.

Clase Arrays: Es una clase de utilidad introducida en el JDK 1.2 que contiene métodos ***static*** para ordenar, llenar, realizar búsquedas y comparar los arrays clásicos del lenguaje. Permite también ver los ***arrays*** como ***lists***.

Después de esta visión general de la *Java Collections Framework*, se verán algunos detalles de las clases e interfaces más importantes.

4.5.4.2 Interface Collection

La interface ***Collection*** es implementada por los ***conjuntos*** (*sets*) y las ***listas*** (*lists*). Esta interface declara una serie de métodos generales utilizables con ***Sets*** y ***Lists***. La declaración o header de dichos métodos se puede ver ejecutando el comando **> *javap java.util.Collection*** en una ventana de MS-DOS. El resultado se muestra a continuación:

```
public interface java.util.Collection
{
    public abstract boolean add(java.lang.Object);              // opcional
    public abstract boolean addAll(java.util.Collection);       // opcional
    public abstract void clear();                               // opcional
    public abstract boolean contains(java.lang.Object);
    public abstract boolean containsAll(java.util.Collection);
    public abstract boolean equals(java.lang.Object);
    public abstract int hashCode();
    public abstract boolean isEmpty();
    public abstract java.util.Iterator iterator();
    public abstract boolean remove(java.lang.Object);           // opcional
    public abstract boolean removeAll(java.util.Collection);    // opcional
    public abstract boolean retainAll(java.util.Collection);    // opcional
    public abstract int size();
    public abstract java.lang.Object toArray()[];
    public abstract java.lang.Object toArray(java.lang.Object[])[];
}
```

A partir del nombre, de los argumentos y del valor de retorno, la mayor parte de estos métodos resultan autoexplicativos. A continuación se introducen algunos comentarios sobre los aspectos que pueden resultar más novedosos de estos métodos. Los detalles se pueden consultar en la documentación de ***Java***.

Los métodos indicados como "// opcional" (estos caracteres han sido introducidos por los autores de este manual) pueden no estar disponibles en algunas implementaciones, como por ejemplo en las clases que no permiten modificar sus objetos. Por supuesto dichos métodos deben ser definidos, pero lo que hacen al ser llamados es lanzar una ***UnsupportedOperationException***.

El método ***add()*** trata de añadir un objeto a una colección, pero puede que no lo consiga si la colección es un ***set*** que ya tiene ese elemento. Devuelve ***true*** si el método ha llegado a modificar la colección. Lo mismo sucede con ***addAll()***. El método ***remove()*** elimina un único elemento (si lo encuentra), y devuelve ***true*** si la colección ha sido modificada.

El método ***iterator()*** devuelve una referencia ***Iterator*** que permite recorrer una colección con los métodos ***next()*** y ***hasNext()***. Permite también borrar el elemento actual con ***remove()***.

Los dos métodos ***toArray()*** permiten convertir una colección en un array.

4.5.4.3 Interfaces Iterator y ListIterator

La interface ***Iterator*** sustituye a ***Enumeration***, utilizada en versiones anteriores del JDK. Dispone de los métodos siguientes:

```
Compiled from Iterator.java
public interface java.util.Iterator
{
    public abstract boolean hasNext();
    public abstract java.lang.Object next();
    public abstract void remove();
}
```

El método ***remove()*** permite borrar el último elemento accedido con ***next()***. Es la única forma segura de eliminar un elemento mientras se está recorriendo una colección.

Los métodos de la interface ***ListIterator*** son los siguientes:

```
Compiled from ListIterator.java
public interface java.util.ListIterator extends java.util.Iterator
{
    public abstract void add(java.lang.Object);
    public abstract boolean hasNext();
    public abstract boolean hasPrevious();
    public abstract java.lang.Object next();
    public abstract int nextIndex();
    public abstract java.lang.Object previous();
    public abstract int previousIndex();
    public abstract void remove();
    public abstract void set(java.lang.Object);
}
```

La interface ***ListIterator*** permite recorrer una lista en ambas direcciones, y hacer algunas modificaciones mientras se recorre. Los elementos se numeran desde 0 a *n-1*, pero los valores válidos para el índice son de 0 a *n*. Puede suponerse que el índice *i* está en la frontera entre los elementos *i-1* e *i*; en ese caso ***previousIndex***() devolvería *i-1* y ***nextIndex()*** devolvería *i*. Si el índice es 0, ***previousIndex()*** devuelve –1 y si el índice es *n* ***nextIndex()*** devuelve el resultado de ***size()***.

4.5.4.4 Interfaces Comparable y Comparator

Estas interfaces están orientadas a mantener ordenadas las ***listas***, y también los ***sets*** y ***maps*** que deben mantener un orden. Para ello se dispone de las interfaces ***java.lang.Comparable*** y ***java.util.Comparator*** (obsérvese que pertenecen a packages diferentes).

La interface ***Comparable*** declara el método ***compareTo()*** de la siguiente forma:

```
public int compareTo(Object obj)
```

que compara su argumento implícito con el que se le pasa por ventana. Este método devuelve un entero ***negativo***, ***cero*** o ***positivo*** según el argumento implícito (***this***) sea ***anterior***, ***igual*** o ***posterior*** al objeto ***obj***. Las listas de objetos de clases que implementan esta interface tienen un ***orden natural***. En ***Java 1.2*** esta interface está implementada -entre otras- por las clases ***String***, ***Character***, ***Date***, ***File***, ***BigDecimal***, ***BigInteger***, ***Byte***, ***Short***, ***Integer***, ***Long***, ***Float*** y ***Double***. Téngase en cuenta que la implementación estándar de estas clases no asegura un orden alfabético correcto con mayúsculas y minúsculas, y tampoco en idiomas distintos del inglés.

Si se redefine, el método ***compareTo()*** debe ser programado con cuidado: es muy conveniente que sea coherente con el método ***equals()*** y que cumpla la ***propiedad transitiva***. Para más información, consultar la documentación del JDK 1.2.

Las ***listas*** y los ***arrays*** cuyos elementos implementan ***Comparable*** pueden ser ordenadas con los métodos static ***Collections.sort()*** y ***Arrays.sort()***.

La interface ***Comparator*** permite ordenar listas y colecciones cuyos objetos pertenecen a clases de tipo cualquiera. Esta interface permitiría por ejemplo ordenar figuras geométricas planas por el área o el perímetro. Su papel es similar al de la interface ***Comparable***, pero el usuario debe

siempre proporcionar una implementación de esta interface. Sus dos métodos se declaran en la forma:

```
public int compare(Object o1, Object o2)
public boolean equals(Object obj)
```

El objetivo del método ***equals()*** es comparar ***Comparators***.

El método ***compare()*** devuelve un entero ***negativo***, ***cero*** o ***positivo*** según su primer argumento sea ***anterior***, ***igual*** o ***posterior*** al segundo. Los objetos que implementan ***Comparator*** pueden pasarse como argumentos al método ***Collections.sort()*** o a algunos ***constructores*** de las clases ***TreeSet*** y ***TreeMap***, con la idea de que las mantengan ordenadas de acuerdo con dicho ***Comparator***. Es muy importante que ***compare()*** sea compatible con el método ***equals()*** de los objetos que hay que mantener ordenados. Su implementación debe cumplir unas condiciones similares a las de ***compareTo()***.

Java 1.2 dispone de clases capaces de ordenar cadenas de texto en diferentes lenguajes. Para ello se puede consultar la documentación sobre las clases ***CollationKey***, ***Collator*** y sus clases derivadas, en el package ***java.text***.

4.5.4.5 Sets y SortedSets

La interface ***Set*** sirve para acceder a una colección sin elementos repetidos. La colección puede estar o no ordenada (con un orden natural o definido por el usuario, se entiende). La interface ***Set*** no declara ningún método adicional a los de ***Collection***.

Como un ***Set*** no admite elementos repetidos es importante saber cuándo dos objetos son considerados iguales (por ejemplo, el usuario puede o no desear que las palabras ***Mesa*** y ***mesa*** sean consideradas iguales). Para ello se dispone de los métodos ***equals()*** y ***hashcode()***, que el usuario puede redefinir si lo desea.

Utilizando los métodos de ***Collection***, los ***Sets*** permiten realizar operaciones algebraicas de ***unión***, ***intersección*** y ***diferencia***. Por ejemplo, ***s1.containsAll(s2)*** permite saber si ***s2*** está contenido en ***s1***; ***s1.addAll(s2)*** permite convertir ***s1*** en la unión de los dos conjuntos; ***s1.retainAll(s2)*** permite convertir ***s1*** en la intersección de ***s1*** y ***s2***; finalmente, ***s1.removeAll(s2)*** convierte ***s1*** en la diferencia entre ***s1*** y ***s2***.

La interface ***SortedSet*** extiende la interface ***Set*** y añade los siguientes métodos:

```
Compiled from SortedSet.java
public interface java.util.SortedSet extends java.util.Set
{
    public abstract java.util.Comparator comparator();
    public abstract java.lang.Object first();
    public abstract java.util.SortedSet headSet(java.lang.Object);
    public abstract java.lang.Object last();
    public abstract java.util.SortedSet subSet(java.lang.Object, java.lang.Object);
    public abstract java.util.SortedSet tailSet(java.lang.Object);
}
```

que están orientados a trabajar con el “orden”. El método ***comparator()*** permite obtener el objeto pasado al constructor para establecer el orden. Si se ha utilizado el orden natural definido por la interface ***Comparable***, este método devuelve ***null***. Los métodos ***first()*** y ***last()*** devuelven el primer y último elemento del conjunto. Los métodos ***headSet()***, ***subSet()*** y ***tailSet()*** sirven para obtener sub-conjuntos al principio, en medio y al final del conjunto original (los dos primeros no incluyen el límite superior especificado).

Existes dos implementaciones de conjuntos: la clase ***HashSet*** implementa la interface ***Set***, mientras que la clase ***TreeSet*** implementa ***SortedSet***. La primera está basada en una hash table y la segunda en un ***TreeMap***.

Los elementos de un ***HashSet*** no mantienen el orden natural, ni el orden de introducción. Los elementos de un ***TreeSet*** mantienen el orden natural o el especificado por la interface ***Comparator***. Ambas clases definen constructores que admiten como argumento un objeto ***Collection***, lo cual permite convertir un ***HashSet*** en un ***TreeSet*** y viceversa.

4.5.4.6 Listas

La interface ***List*** define métodos para operar con colecciones ordenadas y que pueden tener elementos repetidos. Por ello, dicha interface declara métodos adicionales que tienen que ver con el orden y con el acceso a elementos o rangos de elementos. Además de los métodos de ***Collection***, la interface ***List*** declara los métodos siguientes:

```
Compiled from List.java
public interface java.util.List extends java.util.Collection
{
    public abstract void add(int, java.lang.Object);
    public abstract boolean addAll(int, java.util.Collection);
    public abstract java.lang.Object get(int);
    public abstract int indexOf(java.lang.Object);
    public abstract int lastIndexOf(java.lang.Object);
    public abstract java.util.ListIterator listIterator();
    public abstract java.util.ListIterator listIterator(int);
    public abstract java.lang.Object remove(int);
    public abstract java.lang.Object set(int, java.lang.Object);
    public abstract java.util.List subList(int, int);
}
```

Los nuevos métodos ***add()*** y ***addAll()*** tienen un argumento adicional para insertar elementos en una posición determinada, desplazando el elemento que estaba en esa posición y los siguientes. Los métodos ***get()*** y ***set()*** permiten obtener y cambiar el elemento en una posición dada. Los métodos ***indexOf()*** y ***lastIndexOf()*** permiten saber la posición de la primera o la última vez que un elemento aparece en la lista; si el elemento no se encuentra se devuelve –1.

El método ***subList(int fromIndex, toIndex)*** devuelve una "vista" de la lista, desde el elemento ***fromIndex*** inclusive hasta el ***toIndex*** exclusive. Un cambio en esta "vista" se refleja en la lista original, aunque no conviene hacer cambios simultáneamente en ambas. Lo mejor es eliminar la "vista" cuando ya no se necesita.

Existen dos implementaciones de la interface ***List***, que son las clases ***ArrayList*** y ***LinkedList***. La diferencia está en que la primera almacena los elementos de la colección en un ***array*** de ***Objects***, mientras que la segunda los almacena en una ***lista vinculada***. Los ***arrays*** proporcionan una forma de acceder a los elementos mucho más eficiente que las listas vinculadas. Sin embargo tienen dificultades para crecer (hay que reservar memoria nueva, copiar los elementos del array antiguo y liberar la memoria) y para insertar y/o borrar elementos (hay que desplazar en un sentido u en otro los elementos que están detrás del elemento borrado o insertado). Las ***listas vinculadas*** sólo permiten acceso secuencial, pero tienen una gran flexibilidad para crecer, para borrar y para insertar elementos. El optar por una implementación u otra depende del caso concreto de que se trate.

4.5.4.7 Maps y SortedMaps

Un ***Map*** es una estructura de datos agrupados en parejas ***clave/valor***. Pueden ser considerados como una tabla de dos columnas. La ***clave*** debe ser única y se utiliza para acceder al ***valor***.