En esta sección se va a tratar las Colecciones, o Estructuras de Datos, que Java aporta para la manipulación y Almacenamiento de Datos e información. Se presentan en primer lugar las Colecciones que están disponibles en la versión 1.1 del JDK, para luego tratar las nuevas Colecciones que se incorporan al lenguaje con el JDK 1.2 y que representan un tremendo avance en la potencia y facilidad de uso de las estruturas de datos orientadas al almacenamiento de información.

Arrays

Mucho de lo que se podría decir de los arrays ya se cuenta en otra sección, aquí sólo interesa el array como almacén de objetos. Hay dos características que diferencian a los arrays de cualquier otro tipo de colección: eficiencia y tipo. El array es la forma más eficiente que Java proporciona para almacenar y acceder a una secuencia de objetos. El array es una simple secuencia lineal, que hace que el acceso a los elementos sea muy rápido, pero el precio que hay que pagar por esta velocidad es que cuando se crea un array su tamaño es fijado y no se puede cambiar a lo largo de la vida del objeto. Se puede sugerir la creación de un array de tamaño determinado y luego, ya en tiempo de ejecución, crear otro más grande, mover todos los objetos al nuevo y borrar el antiguo. Esto es lo que hace la clase **Vector**, que se verá posteriormente, pero debido a la carga que supone esta flexibilidad, un **Vector** es menos eficiente que un array, en cuestiones de velocidad.

La clase **vector** en C++ no sabe el tipo de los objetos que contiene, pero tiene un inconveniente cuando se la compara con los arrays en Java: el operador [] de la clase **vector** de C++ no realiza un chequeo de límites, así que uno se puede pasar del final (aunque es posible saber el tamaño del vector invocando al método *at()* para realizar la comprobación de límites, si se desea). En Java, independientemente de que se esté utilizando un array o una colección, siempre hay comprobación de límites, y el sistema lanzará una excepción en caso de que se intente el acceso a un elemento que se encuentra fuera de los límites. Así pues, la razón de que C++ no compruebe límites es la velocidad, y con esta sobrecarga hay que vivir siempre en Java, porque todas las veces realizará la comprobación de que no se acceda a un lugar fuera del array o la colección.

Los otros tipos de colecciones disponibles en Java: **Vector**, **Stack** y **Hashtable**; pueden contener cualquier tipo de objeto, sin necesidad de que sea de un tipo definido. Esto es así porque tratan a sus elementos como si fuesen **Object**, la clase raíz de todas las clases Java. Esto es perfecto desde el punto de vista de que se construye solamente una colección, y cualquier tipo de objeto puede ser almacenado en ella. Pero aquí es donde los arrays vuelven a ser más eficientes que las colecciones genéricas, porque cuando se crea un arraya hay que indicar el tipo de objetos que va a contener. Esto significa que ya en tiempo de compilación se realizan comprobaciones para que no se almacene en el array ningún objeto de tipo diferente al que está destinado a contener, ni que se intente extraer un objeto diferente. Desde luego, Java controlará de que no se envíe un mensaje inadecuado a un objeto, ya sea en tiempo de compilación como en tiempo de ejecución.

Así que, tanto por eficiencia como por comprobación de tipos, es mejor utilizar un array siempre que se pueda. Sin embargo, cuando se trata de resolver un problema más general, los arrays pueden ser muy restrictivos y es entonces cuando hay que recurrir a otro tipo de almacenamiento de datos.

Las colecciones de clases manejan solamente los identificadores, *handles*, de los objetos. Un array, sin embargo, puede crearse para contener tipos básicos directamente, o también identificadores de objetos. Es posible utilizar las clases correspondientes a los tipos básicos, como son **Integer**, **Double**, etc., para colocar

tipos básicos dentro de una colección. El colocar una cosa u otra es cuestión de eficiencia, porque es mucho más rápida la creación y acceso en un array de tipos básicos que en uno de objetos del tipo básico.

Desde luego, si se está utilizando un tipo básico y se necesita la flexibilidad que ofrece una colección de expandirse cuando sea preciso, el array no sirve y habrá que recurrir a la colección de objetos del tipo básico. Quizás se podría pensar que un **Vector** especializado en cada uno de los tipos básicos podría ser casi igual de eficiente que un array, pero por desgracia, Java no proporciona mas que un tipo de genérico de **Vector**, en el que se puede meter de todo. Este es otro de las cuestiones que Java tiene pendientes.

Colecciones

Cuando se necesitan características más sofisticadas para almacenar objetos, que las que proporciona un simple array, Java pone a disposición del programador las clases colección: **Vector**, **BitSet**, **Stack** y **Hashtable**.

Entre otras características, las clases colección se redimensionan automáticamente, por lo que se puede colocar en ellas cualquier número de objetos, sin necesidad de tener que ir controlando continuamente en el programa la longitud de la colección.

La gran *desventaja* del uso de las colecciones en Java es que se pierde la información de tipo cuando se coloca un objeto en una colección. Esto ocurre porque cuando se escribió la colección, el programador de esa colección no tenía ni idea del tipo de datos específicos que se iban a colocar en ella, y teniendo en mente el hacer una herramienta lo más general posible, se hizo que manejase directamente objetos de tipo **Object**, que es el objeto raíz de todas las clases en Java. La solución es perfecta, excepto por dos razones:

1. Como la información de tipo se pierde al colocar un objeto en la colección, cualquier tipo de objeto se va a poder colar en ella, es decir, si la colección está destinada a contener animales mamíferos, nada impide que se pueda colar un coche en ella.
2. Por la misma razón de la pérdida de tipo, la única cosa que sabe la colección es que maneja un **Object**. Por ello, hay que colocar siempre un moldeo al tipo adecuado antes de utilizar cualquier objeto contenido en una colección.

La verdad es que no todo es tan negro, Java no permite que se haga uso inadecuado de los objetos que se colocan en una colección. Si se introduce un coche en una colección de animales mamíferos, al intentar extraer el coche se obtendrá una excepción. Y del mismo modo, si se intenta colocar un moldeo al coche que se está sacando de la colección para convertirlo en animal mamífero, también se obtendrá una excepción en tiempo de ejecución. El ejemplo java411.java ilustra estas circunstancias.

```
import java.util.*;

class Coche {
    private int numCoche;
    Coche( int i ) {
        numCoche = i;
        }
    void print() {
        System.out.println( "Coche #"+numCoche );
        }
    }

class Barco {
```

```
    private int numBarco;
    Barco( int i ) {
        numBarco = i;
        }
    void print() {
        System.out.println( "Barco #"+numBarco );
        }
    }

public class java411 {
    public static void main( String args[] ) {
        Vector coches = new Vector();
        for( int i=0; i < 7; i++ )
            coches.addElement( new Coche( i ) );
        // No hay ningun problema en añadir un barco a los coches
        coches.addElement( new Barco( 7 ) );
        for( int i=0; i < coches.size(); i++ )
            (( Coche )coches.elementAt( i ) ).print();
        // El barco solamente es detectado en tiempo de ejecucion
        }
    }
```

Como se puede observar, el uso de un **Vector** es muy sencillo: se crea uno, se colocan elementos en él con el método *addElement()* y se recuperan con el método *elementAt()*. **Vector** tiene el método *size()* que permite conocer cuántos elementos contiene, para evitar el acceso a elementos fuera de los límites del **Vector** y obtener una excepción.

Las clases **Coche** y **Barco** son distintas, no tienen nada en común excepto que ambas son **Object**. Si no se indica explícitamente de la clase que se está heredando, automáticamente se hereda de **Object**. La clase **Vector** maneja elementos de tipo **Object**, así que no solamente es posible colocar en ella objetos **Coche** utilizando el método *addElement()*, sino que también se pueden colocar elementos de tipo **Barco** sin que haya ningún problema ni en tiempo de compilación ni a la hora de ejecutar el programa. Cuando se recupere un objeto que se supone es un **Coche** utilizando el método *elementAt()* de la clase **Vector**, hay que colocar un moldeo para convertir el objeto **Object** en el **Coche** que se espera, luego hay que colocar toda la expresión entre paréntesis para forzar la evaluación del moldeo antes de llamar al método *print()* de la clase **Coche**, sino habrá un error de sintaxis. Posteriormente, ya en tiempo de ejecución, cuando se intente moldear un objeto **Barco** a un **Coche**, se generará una excepción, tal como se puede comprobar en las siguientes líneas, que reproducen la salida de la ejecución del ejemplo:

```
%java java411
Coche #0
Coche #1
Coche #2
Coche #3
Coche #4
Coche #5
Coche #6
java.lang.ClassCastException: Barco
    at java411.main(java411.java:54)
```

Lo cierto es que esto es un fastidio, porque puede ser la fuente de errores que son muy difíciles de encontrar. Si en una parte, o en varias partes, del programa se insertan elementos en la colección, y se descubre en otra parte diferente del programa que se genera una excepción es porque hay algún elemento erróneo en la colección, así que hay que buscar el sitio donde se ha insertado el elemento de la discordia, lo cual puede llevar a intensas sesiones de depuración. Así que, para enredar al principio, es mejor empezar con clases estandarizadas en vez de aventurarse en otras más complicadas, a pesar de que estén menos optimizadas.

Enumeraciones

En cualquier clase de colección, debe haber una forma de meter cosas y otra de sacarlas; después de todo, la principal finalidad de una colección es almacenar

cosas. En un **Vector**, el método *addElement()* es la manera en que se colocan objetos dentro de la colección y llamando al método *elementAt()* es cómo se sacan. **Vector** es muy flexible, se puede seleccionar cualquier cosa en cualquier momento y seleccionar múltiples elementos utilizando diferentes índices.
Si se quiere empezar a pensar desde un nivel más alto, se presenta un inconveniente: la necesidad de saber el tipo exacto de la colección para utilizarla. Esto no parece que sea malo en principio, pero si se empieza implementando un **Vector** a la hora de desarrollar el programa, y posteriormente se decide cambiarlo a **List**, por eficiencia, entonces sí es problemático.
El concepto de *enumerador*, o *iterador*, que es su nombre más común en C++ y OOP, puede utilizarse para alcanzar el nivel de abstracción que se necesita en este caso. Es un objeto cuya misión consiste en moverse a través de una secuencia de objetos y seleccionar aquellos objetos adecuados sin que el programador cliente tenga que conocer la estructura de la secuencia . Además, un iterador es normalmente un objeto ligero, *lightweight*, es decir, que consumen muy pocos recursos, por lo que hay ocasiones en que presentan ciertas restricciones; por ejemplo, algunos iteradores solamente se puede mover en una dirección.
La **Enumeration** en Java es un ejemplo de un iterador con esas características, y las cosas que se pueden hacer son:

- Crear una colección para manejar una **Enumeration** utilizando el método *elements()*. Esta **Enumeration** estará lista para devolver el primer elemento en la secuencia cuando se llame por primera vez al método *nextElement()*.
- Obtener el siguiente elemento en la secuencia a través del método *nextElement()*.
- Ver si hay más elementos en la secuencia con el método *hasMoreElements()*.

Y esto es todo. No obstante, a pesar de su simplicidad, alberga bastante poder. Para ver cómo funciona, el ejemplo java412.java, es la modificación de anterior, en que se utilizaba el método *elementAt()* para seleccionar cada uno de los elementos. Ahora se utiliza una enumeración para el mismo propósito, y el único código interesante de este nuevo ejemplo es el cambio de las líneas del ejemplo original

```
for( int i=0; i < coches.size(); i++ )
    (( Coche )coches.elementAt( i ) ).print();
```

por estas otras en que se utiliza la enumeración para recorrer la secuencia de objetos

```
while( e.hasMoreElements() )
    (( Coche )e.nextElement()).print();
```

Con la **Enumeration** no hay que preocuparse del número de elementos que contenga la colección, ya que del control sobre ellos se encargan los métodos *hasMoreElements()* y *nextElement()*.

Tipos de Colecciones

Con el JDK 1.0 y 1.1 se proporcionaban librerías de colecciones muy básicas, aunque suficientes para la mayoría de los proyectos. En el JDK 1.2 ya se amplía esto y, además, las anteriores colecciones han sufrido un profundo rediseño. A continuación se verán cada una de ellas por separado para dar una idea del potencial que se ha incorporado a Java.

Vector

El **Vector** es muy simple y fácil de utilizar. Aunque los métodos más habituales en su manipulación son *addElement()* para insertar elementos en el **Vector**, *elementAt()* para recuperarlos y *elements()* para obtener una **Enumeration** con el número de elementos del **Vector**, lo cierto es que hay más métodos, pero no es el momento de relacionarlos todos, así que, al igual que sucede con todas las librerías

de Java, se remite al lector a que consulte la documentación electrónica que proporciona Javasoft, para conocer los demás métodos que componen esta clase.
Las colecciones estándar de Java contienen el método *toString()*, que permite obtener una representación en forma de **String** de sí mismas, incluyendo los objetos que contienen. Dentro de **Vector**, por ejemplo, *toString()* va saltando a través de los elementos del **Vector** y llama al método *toString()* para cada uno de esos elementos. En caso, por poner un ejemplo, de querer imprimir la dirección de la clase, parecería lógico referirse a ella simplemente como **this** (los programadores C++ estarán muy inclinados a esta posibilidad), así que tendríamos el código que muestra el ejemplo java413.java y que se reproduce en las siguientes líneas.

```
import java.util.*;

public class java413 {
    public String toString() {
        return( "Direccion del objeto: "+this+"\n" );
        }

    public static void main( String args[] ) {
        Vector v = new Vector();

        for( int i=0; i < 10; i++ )
            v.addElement( new java413() );
        System.out.println( v );
        }
    }
```

El ejemplo no puede ser más sencillo, simplemente crea un objeto de tipo **java413** y lo imprime; sin embargo, a la hora de ejecutar el programa lo que se obtiene es una secuencia infinita de excepciones. Lo que está pasando es que cuando se le indica al compilador:

```
"Direccion del objeto: "+this
```

el compilador ve un **String** seguido del operador + y otra cosa que no es un **String**, así que intenta convertir **this** en un **String**. La conversión la realiza llamando al método *toString()* que genera una llamada recursiva, llegando a llenarse la pila.
Si realmente se quiere imprimir la dirección del objeto en este caso, la solución pasa por llamar al método *toString()* de la clase **Object**. Así, si en vez de **this** se coloca *super.toString()*, el ejemplo funcionará. En otros casos, este método también funcionará siempre que se esté heredando directamente de **Object** o, aunque no sea así, siempre que ninguna clase padre haya sobreescrito el método *toString()*.

BitSet

Se llama así lo que en realidad es un **Vector** de bits. Lo que ocurre es que está optimizado para uso de bits. Bueno, optimizado en cuanto a tamaño, porque en lo que respecta al tiempo de acceso a los elementos, es bastante más lento que el acceso a un array de elementos del mismo tipo básico.
Además, el tamaño mínimo de un **BitSet** es de 64 bits. Es decir, que si se está almacenando cualquier otra cosa menor, por ejemplo de 8 bits, se estará desperdiciando espacio.
En un **Vector** normal, la colección se expande cuando se añaden más elementos. En el **BitSet** ocurre los mismo pero ordenadamente. El ejemplo java414.java, muestra el uso de esta colección.
Se utiliza el generador de números aleatorios para obtener un `byte`, un `short` y un `int`, que son convertidos a su patrón de bits e incorporados al **BitSet**.

Stack

Un **Stack** es una ***Pila***, o una colección de tipo *LIFO* (*last-in*, *first-out*). Es decir, lo último que se coloque en la pila será lo primero que se saque. Como en todas las colecciones de Java, los elementos que se introducen y sacan de la pila son **Object**, así que hay que tener cuidado con el moldeo a la hora de sacar alguno de ellos.
Los diseñadores de Java, en vez de utilizar un **Vector** como bloque para crear un **Stack**, han hecho que **Stack** derive directamente de **Vector**, así que tiene todas las características de un **Vector** más alguna otra propia ya del **Stack**. El ejemplo

siguiente, java415.java, es una demostración muy simple del uso de una ***Pila*** que consisten en leer cada una de las líneas de un array y colocarlas en un **String**. Cada línea en el array `diasSemana` se inserta en el **Stack** con *push()* y posteriormente se retira con *pop()*. Para ilustrar una afirmación anterior, también se utilizan métodos propios de **Vector** sobre el **Stack**. Esto es posible ya que en virtud de la herencia un **Stack** es un **Vector**, así que todas las operaciones que se realicen sobre un **Vector** también se podrán realizar sobre un **Stack**, como por ejemplo, *elementAt()*.

Hashtable

Un **Vector** permite selecciones desde una colección de objetos utilizando un número, luego parece lógico pensar que hay números asociados a los objetos. Bien, entonces ¿qué es lo que sucede cuando se realizan selecciones utilizando otros criterios? Un **Stack** podría servir de ejemplo: su criterio de selección es "lo último que se haya colocado en el **Stack**". Si rizamos la idea de "selección desde una secuencia", nos encontramos con un *mapa*, un *diccionario* o un *array asociativo*. Conceptualmente, todo parece ser un vector, pero en lugar de acceder a los objetos a través de un número, en realidad se utiliza *otro objeto*. Esto nos lleva a utilizar *claves* y al procesado de claves en el programa. Este concepto se expresa en Java a través de la clase abstracta **Dictionary**. El interfaz para esta clase es muy simple:

- *size()*, indica cuántos elementos contiene,
- *isEmpty()*, es **true** si no hay ningún elemento,
- *put(Object clave,Object valor)*, añade un **valor** y lo asocia con una **clave**
- *get(Object clave)*, obtiene el **valor** que corresponde a la **clave** que se indica
- *remove(Object clave)*, elimina el par **clave-valor** de la lista
- *keys()*, genera una **Enumeration** de todas las claves de la lista
- *elements()*, genera una **Enumeration** de todos los valores de la lista

Todo es lo que corresponde a un *Diccionario* (**Dictionary**), que no es excesivamente difícil de implementar. El ejemplo java416.java es una aproximación muy simple que utiliza dos **Vectores**, uno para las claves y otro para los valores que corresponden a esas claves.

```
import java.util.*;

public class java416 extends Dictionary {
    private Vector claves = new Vector();
    private Vector valores = new Vector();
    public int size() {
        return( claves.size() );
        }
    public boolean isEmpty() {
        return( claves.isEmpty() );
        }

    public Object put( Object clave,Object valor ) {
        claves.addElement( clave );
        valores.addElement( valor );
        return( clave );
        }

    public Object get( Object clave ) {
        int indice = claves.indexOf( clave );
        // El metodo indexOf() devuelve -1 si no encuentra la clave
que se
        // esta buscando
        if( indice == -1 )
            return( null );
        return( valores.elementAt( indice ) );
        }

    public Object remove(Object clave) {
```

```
        int indice = claves.indexOf( clave );

        if( indice == -1 )
            return( null );
        claves.removeElementAt( indice );
        Object valorRetorno = valores.elementAt( indice );
        valores.removeElementAt( indice );
        return( valorRetorno );
        }

    public Enumeration keys() {
        return( claves.elements() );
        }

    public Enumeration elements() {
        return( valores.elements() );
        }

    // Ahora es cuando se prueba el ejemplo
    public static void main( String args[] ) {
        java416 ej = new java416();
        for( char c='a'; c <= 'z'; c++ )
            ej.put( String.valueOf( c ),String.valueOf( c
).toUpperCase() );

        char[] vocales = { 'a','e','i','o','u' };
        for( int i=0; i < vocales.length; i++ )
            System.out.println( "Mayusculas: " +
            ej.get( String.valueOf( vocales[i] ) ) );
        }
    }
```

La primera cosa interesante que se puede observar en la definición de **java416** es que *extiende* a **Dictionary**. Esto significa que **java416** es *un tipo de* **Diccionario**, con lo cual se pueden realizar las mismas peticiones y llamar a los mismos métodos que a un **Diccionario**. A la hora de construirse un **Diccionario** propio todo lo que se necesita es rellenar todos los métodos que hay en **Dictionary**. Se deben sobreescribir todos ellos, excepto el constructor, porque todos son abstractos.
Los **Vectores** `claves` y `valores` están relacionados a través de un número índice común. Es decir, si se llama al método *put()* con la clave "león" y el valor "rugido" en la asociación de animales con el sonido que producen, y ya hay 100 elementos en la clase **java416**, entonces "león" será el elemento 101 de `claves` y "rugido" será el elemento 101 de `valores`. Y cuando se pasa al método *get()* como parámetro "león", genera el número índice con *claves.indexOf()*, y luego utiliza este índice para obtener el valor asociado en el vector `valores`.
Para mostrar el funcionamiento, en *main()* se utiliza algo tan simple como mapear las letras minúsculas y mayúsculas, que aunque se pueda hacer de otras formas más eficientes, sí sirve para mostrar el funcionamiento de la clase, que es lo que se pretende por ahora.
La librería estándar de Java solamente incorpora una implementación de un **Dictionary**, la **Hashtable**. Esta **Hashtable** tiene el mismo interfaz básico que la clase del ejemplo anterior **java416**, ya que ambas heredan de **Dictionary**, pero difiere en algo muy importante: la *eficiencia*. Si en un **Diccionario** se realiza un *get()* para obtener un valor, se puede observar que la búsqueda es bastante lenta a través del vector de claves. Aquí es donde la **Hashtable** acelera el proceso, ya que en vez de realizar la tediosa búsqueda línea a línea a través del vector de claves, utiliza un valor especial llamado *código hash*. El *código hash* es una forma de conseguir información sobre el objeto en cuestión y convertirlo en un `int` relativamente único para ese objeto. Todos los objetos tienen un *código hash* y *hashCode()* es un método de la clase **Object**. Una **Hashtable** coge el *hashCode()* del objeto y lo utiliza para cazar rápidamente la clave. El resultado es una impresionante reducción del tiempo de búsqueda. La forma en que funciona una tabla **Hash** se escapa del Tutorial, hay muchos libros que lo explican en detalle, por

ahora es suficiente con saber que la tabla **Hash** es un **Diccionario** muy rápido y que un **Diccionario** es una herramienta muy útil.
Para ver el funcionamiento de la tabla **Hash** está el ejemplo java417.java, que intenta comprobar la aleatoriedad del método *Math.random()*. Idealmente, debería producir una distribución perfecta de números aleatorios, pero para poder comprobarlo sería necesario generar una buena cantidad de números aleatorios y comprobar los rangos en que caen. Una **Hashtable** es perfecta para este propósito al asociar objetos con objetos, en este caso, los valores producidos por el método *Math.random()* con el número de veces en que aparecen esos valores.
En el método *main()* del ejemplo, cada vez que se genera un número aleatorio, se convierte en objeto **Integer** para que pueda ser manejado por la tabla **Hash**, ya que no se pueden utilizar tipos básicos con una colección, porque solamente manejan objetos. El método *containsKey()* comprueba si la clave se encuentra ya en la colección. En caso afirmativo, el método *get()* obtiene el valor asociado a la clave, que es un objeto de tipo **Contador**. El valor `i` dentro del contador se incrementa para indicar que el número aleatorio ha aparecido una vez más.
Si la clave no se encuentra en la colección, el método *put()* colocará el nuevo par clave-valor en la tabla **Hash**. Como Contador inicializa automáticamente su variable `i` a 1 en el momento de crearla, ya se indica que es la primera vez que aparece ese número aleatorio concreto.
Para presentar los valores de la tabla **Hash**, simplemente se imprimen. El método *toString()* de **Hashtable** navega a través de los pares clave-valor y llama a método *toString()* de cada uno de ellos. El método *toString()* de **Integer** está predefinido, por lo que no hay ningún problema en llamar a *toString()* para **Contador**. Un ejemplo de ejecución del programa sería la salida que se muestra a continuación:

```
%java java417
{19=526, 18=533, 17=460, 16=513, 15=521, 14=495, 13=512, 12=483,
11=488, 10=487, 9=514, 8=523, 7=497, 6=487, 5=489, 3=509, 2=503,
1=475, 0=505}
```

Al lector le puede parecer superfluo el uso de la clase **Contador**, que parece que no hace nada que no haga ya la clase **Integer**. ¿Por qué no utilizar `int` o **Integer**? Pues bien, `int` no puede utilizarse porque como ya se ha indicado antes, las colecciones solamente manejan objetos, por ello están las clases que envuelven a esos tipos básicos y los convierten en objetos. Sin embargo, la única cosa que pueden hacer estas clases es inicializar los objetos a un valor determinado y leer ese valor. Es decir, no hay modo alguno de cambiar el valor de un objeto correspondiente a un tipo básico, una vez que se ha creado. Esto hace que la clase **Integer** sea inútil para resolver el problema que plantea el ejemplo, así que la creación de la clase **Contador** es imprescindible. Quizás ahora que el lector sabe que no puede colocar objetos creados a partir de las clases correspondientes a tipos básicos en colecciones, estas clases tengan un poco menos de valor, pero... la vida es así, por un lado da y por otro quita... y Java no va a ser algo diferente.
En el ejemplo se utiliza la clase **Integer**, que forma parte de la librería estándar de Java como clave para la tabla **Hash**, y funciona perfectamente porque tiene todo lo necesario para funcionar como clave. Pero un error muy común se presenta a la hora de crear clases propias para que funcionen como claves. Por ejemplo, supóngase que se quiere implementar un sistema de predicción del tiempo en base a objetos de tipo **Oso** y tipo **Prediccion**, para detectar cuando entra la primavera. Tal como se muestra en el ejemplo java418.java, la cosa parece muy sencilla, se crean las dos clases y se utiliza **Oso** como clave y **Prediccion** como valor.
Cada **Oso** tendrá un número de identificación, por lo que sería factible buscar una **Prediccion** en la tabla **Hash** de la forma: "*Dime la **Prediccion** asociada con el **Oso** número 3*". La clase **Prediccion** contiene un booleano que es inicializado utilizando *Math.random()*, y una llamada al método *toString()* convierte el resultado en algo legible. En el método *main()*, se rellena una **Hashtable** con los **Osos** y sus **Predicciones** asociadas. Cuando la tabla **Hash** está completa, se imprime. Y ya se hace la consulta anterior sobre la tabla para buscar la **Prediccion** que corresponde al **Oso** número 3.

Esto parece simple y suficiente, pero no funciona. El problema es que **Oso** deriva directamente de la clase raíz **Object**, que es lo que ocurre cuando no se especifica una clase base, que en última instancia se hereda de **Object**. Luego es el método *hashCode()* de **Object** el que se utiliza para generar el código *hash* para cada objeto que, por defecto, utiliza la dirección de ese objeto. Así, la primera instancia de **Oso(3)** no va a producir un código *hash* igual que producirá una segunda instancia de **Oso(3)**, con lo cual no se puede utilizar para obtener buenos resultados de la tabla.
Se puede seguir pensando con filosofía ahorrativa y decir que todo lo que se necesita es sobreescribir el método *hashCode()* de la forma adecuada y ya está. Pero, esto tampoco va a funcionar hasta que se haga una cosa más: sobreescribir el método *equals()*, que también es parte de **Object**. Este es el método que utiliza la tabla **Hash** para determinar si la clave que se busca es igual a alguna de las claves que hay en la tabla. De nuevo, el método *Object.equals()* solamente compara direcciones de objetos, por lo que un **Oso(3)** probablemente no sea igual a otro **Oso(3)**.
Por lo tanto, a la hora de escribir clases propias que vayan a funcionar como clave en una **Hastable**, hay que sobreescribir los métodos *hashCode()* y *equals()*. El ejemplo java419.java ya se incorporan estas circunstancias.

```
import java.util.*;

// Si se crea una clase que utilice una clave en una Tabla Hash, es
// imprescindible sobreescribir los metodos hashCode() y equals()
// Utilizamos un oso para saber si está hibernando en su temporada de
// invierno o si ya tine que despertarse porque le llega la primavera
class Oso2 {
    int numero;
    Oso2( int n ) {
        numero = n;
        }

    public int hashCode() {
        return( numero );
        }

    public boolean equals( Object obj ) {
        if( (obj != null) && (obj instanceof Oso2) )
            return( numero == ((Oso2)obj).numero );
        else
            return( false );
        }
    }

// En función de la oscuridad, o claridad del día, pues intenta saber
si
// ya ha la primavera ha asomado a nuestras puertas
class Prediccion {
    boolean oscuridad = Math.random() > 0.5;

    public String toString() {
        if( oscuridad )
            return( "Seis semanas mas de Invierno!" );
        else
            return( "Entrando en la Primavera!" );
        }
    }

public class java419 {
    public static void main(String args[]) {
        Hashtable ht = new Hashtable();

        for( int i=0; i < 10; i++ )
            ht.put( new Oso2( i ),new Prediccion() );
        System.out.println( "ht = "+ht+"\n" );
```

```
        System.out.println( "Comprobando la prediccion para el oso
#3:");
        Oso2 oso = new Oso2( 3 );
        if( ht.containsKey( oso ) )
            System.out.println( (Prediccion)ht.get( oso ) );
        }
    }
```

El método *hashCode()* devuelve el número que corresponde a un **Oso** como un identificador, siendo el programador el responsable de que no haya dos números iguales. El método *hashCode()* no es necesario que devuelva un identificador, sino que eso es necesario porque *equals()* debe ser capaz de determinar estrictamente cuando dos objetos son equivalentes.

El método *equals()* realiza dos comprobaciones adicionales, una para comprobar si el objeto es `null`, y, en caso de que no lo sea, comprobar que sea una instancia de **Oso**, para poder realizar las comparaciones, que se basan en los números asignados a cada objeto **Oso**. Cuando se ejecuta este nuevo programa, sí se produce la salida correcta. Hay muchas clases de la librería de Java que sobreescriben los métodos *hashCode()* y *equals()* basándose en el tipo de objetos que son capaces de crear.

Las tablas **Hash** son utilizadas también por muchas clases de la librería estándar de Java, por ejemplo, para obtener las propiedades del sistema se usa la clase **Properties** que hereda directamente de **Hashtable**. Y además, contiene una segunda **Hashtable** en donde guarda las propiedades del sistema que se usan *por defecto*.