9: Guardar objetos

Es un programa bastante simple que sólo tiene una cantidad fija de objetos cuyos periodos de vida son conocidos.

En general, los programas siempre estarán creando nuevos objetos, en base a algún criterio que sólo se conocerá en tiempo de ejecución. No se sabrá hasta ese momento la cantidad o incluso el tipo exacto de objetos necesarios. Para solucionar el problema de programación general, hay que crear cualquier número de objetos, en cualquier momento, en cualquier sitio. Por tanto, no se puede confiar en crear una referencia con nombre que guarde cada objeto:

```
MiObjeto miReferencia;
```

dado que, de hecho, nunca se sabrá cuántas se necesitarán.

Para solucionar este problema tan esencial, Java tiene distintas maneras de guardar los objetos (o mejor, referencias a objetos). El tipo incrustado es el array, lo cual ya hemos discutido anteriormente. Además, la biblioteca de utilidades de Java tiene un conjunto razonablemente completo de *clases contenedoras* (conocidas también como *clases colección*, pero dado que las bibliotecas de Java 2 usan el nombre **Collection** para hacer referencia a un subconjunto particular de la biblioteca, usaremos el término más genérico "contenedor"). Los contenedores proporcionan formas sofisticadas de guardar e incluso manipular objetos.

Arrays

La mayoría de la introducción necesaria a los arrays se encuentra en la última sección del Capítulo 4, que mostraba cómo definir e inicializar un array. El propósito de este capítulo es el almacenamiento de objetos, y un array es justo una manera de guardar objetos. Pero hay muchas otras formas de guardar objetos, así que, ¿qué hace que un array sea tan especial?

Hay dos aspectos que distinguen a los arrays de otros tipos de contenedores: la eficiencia y el tipo. El array es la forma más eficiente que proporciona Java para almacenar y acceder al azar a una secuencia de objetos (verdaderamente, referencias a objeto). El array es una secuencia lineal simple, que hace rápidos los accesos a elementos, pero se paga por esta velocidad: cuando se crea un objeto array, su tamaño es limitado y no puede variarse durante la vida de ese objeto array. Se podría sugerir crear un array de un tamaño particular y, después, si se acaba el espacio, crear uno nuevo y mover todas las referencias del viejo al nuevo. Éste es el comportamiento de la clase **ArrayList**, que será estudiada más adelante en este capítulo. Sin embargo, debido a la sobrecarga de esta flexibilidad de tamaño, un **ArrayList** es mucho menos eficiente que un array.

La clase contenedora **Vector** de C++ *no* conoce el tipo de objetos que guarda, pero tiene un inconveniente diferente cuando se compara con los arrays de Java: el **operador[]** de **vector** de C++ no hace comprobación de límites, por lo que se puede ir más allá de su final[1]. En Java, hay comprobación de límites independientemente de si se está usando un array o un contenedor —se obtendrá una **excepción en tiempo de ejecución** si se exceden los límites. Como se aprenderá en el Capítulo 10, este tipo de excepción indica un error del programador, y por tanto, no es necesario comprobarlo en el código. Aparte de esto, la razón por la que el **vector** de C++ no comprueba los límites en todos los accesos es la velocidad —en Java se tiene sobrecarga de rendimiento constante de comprobación de límites todo el tiempo, tanto para arrays, como para contenedores.

Las otras clases contenedoras genéricas **List**, **Set** y **Map** que se estudiarán más adelante en este capítulo, manipulan los objetos como si no tuvieran tipo específico. Es decir, las tratan como de tipo **Object**, la clase raíz de todas las clases de Java. Esto trabaja bien desde un punto de vista: es necesario construir sólo un contenedor, y cualquier objeto Java entrará en ese contenedor. (Excepto por los tipos primitivos —que pueden ser ubicados en contenedores como constantes, utilizando las clases envolturas primitivas de Java, o como valores modificables envolviendo la propia clase.) Éste es el segundo lugar en el que un array es superior a los contenedores genéricos: cuando se crea un array, se crea para guardar un tipo específico. Esto significa que se da una comprobación de tipos en tiempo de compilación para evitar introducir el tipo erróneo o confundir el tipo que se espera. Por supuesto, Java evitará que se envíe el mensaje inapropiado a un objeto, bien en tiempo de compilación bien en tiempo de ejecución. Por tanto, no supone un riesgo mayor de una o de otra forma, sino que es simplemente más elegante que sea el compilador el que señale el error, además de ser más rápido en tiempo de ejecución, y así habrá menos probabilidades de sorprender al usuario final con una excepción.

En aras de la eficiencia y de la comprobación de tipos, siempre merece la pena intentar usar un array si se puede. Sin embargo, cuando se intenta solucionar un problema más genérico, los arrays pueden ser demasiado restrictivos. Después de ver los arrays, el resto de este capítulo se dedicará a las clases contenedoras proporcionadas por Java.

Los arrays son objetos de primera clase

Independientemente del tipo de array con el que se esté trabajando, el identificador de array es, de hecho, una referencia a un objeto verdadero que se crea en el montículo. Éste es el objeto que mantiene las referencias a los otros objetos, y puede crearse implícitamente como parte de la sintaxis de inicialización del atributo, o explícitamente mediante una sentencia **new**. Parte del objeto array (de hecho, el único atributo o método al que se puede acceder) es el miembro **length** de sólo lectura que dice cuántos elementos pueden almacenarse en ese array objeto. La sintaxis '**[]**' es el otro acceso que se tiene al array objeto.

---

[1] Es posible, sin embargo, preguntar por el tamaño del **vector**, y el método **at( )** *sí* que hará comprobación de límites.

El ejemplo siguiente muestra las distintas formas de inicializar un array, y cómo se pueden asignar las referencias array a distintos objetos array. También muestra que los arrays de objetos y los arrays de tipos primitivos son casi idénticos en uso. La única diferencia es que los arrays de objetos almacenan referencias, mientras que los arrays de primitivas guardan los valores primitivos directamente.

```
//: c09:TerminoArray.java
// Inicialización y reasignación de arrays.

class Mitologia {} // Una pequeña criatura mítica

public class TerminoArray {
  public static void main(String[] args) {
    // Arrays de objetos:
    Mitologia[] a; // Referencia Null
    Mitologia[] b = new Mitologia[5]; // Referencias Null
    Mitologia[] c = new Mitologia[4];
    for(int i = 0; i < c.length; i++)
      c[i] = new Mitologia();
    // Inicialización de agregados:
    Mitologia[] d = {
      new Mitologia(), new Mitologia(), new Mitologia()
    };
    // Inicialización dinámica de agregados:
    a = new Mitologia[] {
      new Mitologia(), new Mitologia()
    };
    System.out.println("a.length=" + a.length);
    System.out.println("b.length = " + b.length);
    // Las referencias internas del array se
    // inicializan automáticamente a null:
    for(int i = 0; i < b.length; i++)
      System.out.println("b[" + i + "]=" + b[i]);
    System.out.println("c.length = " + c.length);
    System.out.println("d.length = " + d.length);
    a = d;
    System.out.println("a.length = " + a.length);

    // Arrays de datos primitivos:
    int[] e; // Referencia null
    int[] f = new int[5];
    int[] g = new int[4];
    for(int i = 0; i < g.length; i++)
      g[i] = i*i;
    int[] h = { 11, 47, 93 };
```

```
    // Error de compilación: variable e sin inicializar:
    //!System.out.println("e.length=" + e.length);
    System.out.println("f.length = " + f.length);
    // Los datos primitivos de dentro del array se
    // inicializan automáticamente a cero:
    for(int i = 0; i < f.length; i++)
      System.out.println("f[" + i + "]=" + f[i]);
    System.out.println("g.length = " + g.length);
    System.out.println("h.length = " + h.length);
    e = h;
    System.out.println("e.length = " + e.length);
    e = new int[] { 1, 2 };
    System.out.println("e.length = " + e.length);
  }
} ///:~
```

He aquí la salida del programa:

```
b.length = 5
b[0]=null
b[1]=null
b[2]=null
b[3]=null
b[4]=null
c.length = 4
d.length = 3
a.length = 3
a.length = 2
f.length = 5
f[0]=0
f[1]=0
f[2]=0
f[3]=0
f[4]=0
g.length = 4
h.length = 3
e.length = 3
e.length = 2
```

Inicialmente una simple referencia **null** (traducción errónea), y el compilador evita que se haga nada con ella hasta que se haya inicializado adecuadamente. El array **b** se inicializa a un array de referencias **Mitología**, pero, de hecho, no se colocan objetos **Mitología** en ese array. Sin embargo, se sigue pudiendo preguntar por el tamaño del array, dado que **b** apuntó a un objeto legítimo. Esto presenta un pequeño inconveniente: no se puede averiguar cuántos elementos hay *en* el array, puesto que **length** dice sólo cuántos elementos *se pueden* ubicar en el array; es decir, el tamaño del objeto array, no el número de objetos que alberga. Sin embargo, cuando se crea un objeto array sus

referencias se inicializan automáticamente a **null**, por lo que se puede ver si una posición concreta del array tiene un objeto, comprobando si es o no **null**. De forma análoga, un array de tipos primitivos se inicializa automáticamente a cero en el caso de los números, **(char)0** en el caso de **caracteres**, y **false** si se trata de **lógicos**.

El array **c** muestra la creación del objeto array seguida de la asignación de objetos **Mitología** a las posiciones del mismo. El array **d** muestra la sintaxis de "inicialización de agregados" que hace que se cree el objeto array (implícitamente en el montículo, con **new**, al igual que ocurre con el array **c**) *y* que se inicialice con objetos **Mitología**, todo ello en una sentencia.

La siguiente inicialización del array podría definirse como "inicialización dinámica de agregados". La inicialización de agregados usada por **d** debe usarse al definir **d**, pero con la segunda sintaxis se puede crear e inicializar un objeto array en cualquier lugar. Por ejemplo, supóngase que **esconder( )** sea un método que toma un array de objetos **Mitología**. Se podría invocar diciendo:

```
esconder(d);
```

pero también se puede crear dinámicamente el array al que se desea pasar el parámetro:

```
esconder(new Mitologia[] ) { new Mitologia (), new Mitologia() });
```

En algunas situaciones, esta nueva sintaxis proporciona una forma más adecuada de escribir código.

La expresión

```
a = d;
```

muestra cómo se puede tomar una referencia adjuntada a un objeto array y asignársela a otro objeto array, exactamente igual que se puede hacer con cualquier otro tipo de referencia a objeto. Ahora tanto **a** como **b** apuntan al mismo objeto array del montículo.

La segunda parte de **TamanioArray.java** muestra que los arrays de tipos primitivos funcionan exactamente igual que los arrays de objetos *excepto* que los arrays de tipos primitivos guardan los valores directamente.

Contendores de datos primitivos

Las clases contenedoras sólo pueden almacenar referencias a objetos. Sin embargo, se puede crear un array para albergar directamente tipos primitivos, al igual que referencias a objetos. *Es* posible utilizar las clases envoltorio ("wrapper") como **Integer**, **Double**, etc. para ubicar valores primitivos dentro de un contenedor, pero estas clases pueden ser tediosas de usar. Además, es mucho más eficiente crear y acceder a un array de datos primitivos que a un contenedor de objetos envoltorio.

Por supuesto, si se está usando un tipo primitivo y se necesita la flexibilidad de un contenedor que se expanda automáticamente cuando se necesita más espacio, el array no será suficiente, por lo que uno se verá obligado a usar un contenedor de objetos envoltorio. Se podría pensar que debería haber un tipo especializado de **ArrayList** por cada tipo de dato primitivo, pero Java no proporciona

esto. Quizás algún día cualquier tipo de mecanismo plantilla proporcione un método que haga que Java maneje mejor este problema[2].

Devolver un array

Suponga que se está escribiendo un método y no se quiere que devuelva sólo un elemento, sino un conjunto de elementos. Los lenguajes como C y C++ hacen que esto sea difícil porque no se puede devolver un array sin más, hay que devolver un puntero a un array. Esto supone problemas pues se vuelve complicado intentar controlar la vida del array, lo que casi siempre acaba llevando a problemas de memoria.

Java sigue un enfoque similar, pero simplemente "devuelve un array". Por supuesto que, de hecho, se está devolviendo una referencia a un array, pero con Java uno nunca tiene por qué preocuparse de ese array —estará por ahí mientras se necesite, y el recolector de basura no lo eliminará hasta que deje de utilizarse.

Como ejemplo, considere que se devuelve un array de **cadenas de caracteres**:

```
//: c09:Helado.java
// Métodos que devuelven arrays.

public class Helado {
  static String[] sabor = {
    "Chocolate", "Fresa",
    "Vainilla", "Menta",
    "Moca y almendras", "Ron con pasas",
    "Praline", "Turrón"
  };
  static String[] ConjuntoSabores(int n) {
    // Forzar a que sea positivo y dentro de los límites:
    n = Math.abs(n) % (sabor.length + 1);
    String[] resultados = new String[n];
    boolean[] seleccionado =
      new boolean[sabor.length];
    for (int i = 0; i < n; i++) {
      int t;
      do
        t = (int)(Math.random() * sabor.length);
      while (seleccionado[t]);
      resultados[i] = sabor[t];
      seleccionado[t] = true;
    }
```

[2] Éste es uno de los puntos en los que C++ es enormemente superior a Java, dado que C++ soporta los *tipos parametrizados* haciendo uso de la palabra clave **template**.

```
    return resultados;
  }
  public static void main(String[] args) {
    for(int i = 0; i < 20; i++) {
      System.out.println(
        "ConjuntoSabores(" + i + ") = ");
      String[] fl = ConjuntoSabores(sabor.length);
      for(int j = 0; j < fl.length; j++)
        System.out.println("\t" + fl[j]);
    }
  }
} ///:~
```

El método **ConjuntoSabores( )** crea un array de **cadenas de caracteres** llamado **resultados**. El tamaño del array es **n**, determinado por el parámetro que se le pasa al método. Posteriormente, procede a elegir sabores de manera aleatoria a partir del array **sabores** y a ubicarlos en **resultados**, que es lo que finalmente devuelve. Devolver el array es exactamente igual que devolver cualquier otro objeto —es una referencia. No es importante en este momento el que el array se haya creado dentro de **ConjuntoSabores( )**, o que el array se haya creado en cualquier otro sitio. El recolector de basura se encarga de limpiar el array una vez que se ha acabado con él, pero mientras tanto, éste seguirá vivo.

Aparte de lo ya comentado, nótese que **ConjuntoSabores( )** elige sabores al azar, asegurando para cada una de las elecciones que ésta no ha salido antes. Esto se hace en un bucle **do** que se encarga de hacer selecciones al azar hasta encontrar una que ya no está en el array **seleccionado**. (Por supuesto, también se podría realizar una comparación de **cadenas de caracteres** para ver si la selección hecha al azar ya estaba en el array **resultados**, pero las comparaciones de **cadenas de caracteres** son ineficientes.) Si tiene éxito, añade la entrada y pasa al siguiente (se incrementa **i**).

El método **main( )** imprime 20 conjuntos completos de sabores, por lo que se puede ver que **ConjuntoSabores( )** elige los sabores en orden aleatorio cada vez. Esto se ve mejor si se redirecciona la salida a un archivo. Y al recorrer el archivo, recuérdese que uno simplemente *quiere* el helado, no lo *necesita*.

La clase **Arrays**

En **java.util** se encuentra la clase **Arrays**, capaz de mantener un conjunto de métodos **estáticos** que llevan a cabo funciones de utilidad para arrays. Tiene cuatro funciones básicas: **equals( )** para comparar la igualdad de dos arrays; **fill( )** para rellenar un array con un valor; **sort( )** para ordenar el array; y **binarySearch( )** para encontrar un dato en un array ordenado. Todos estos métodos están sobrecargados para todos los tipos de datos primitivos y **objetos**. Además, hay un método simple **asList( )** que hace que un array se convierta en un contendor **List**, del cual se aprenderá más adelante en este capítulo.

A la vez que útil, la clase **Arrays** puede dejar de ser completamente funcional. Por ejemplo, sería bueno ser capaces de imprimir los elementos de un array sin tener que codificar el código **for** a mano cada vez. Como se verá, el método **fill( )** sólo toma un único valor y lo posiciona en el array, por lo que si se deseaba —por ejemplo— rellenar un array con números generados al azar, **fill( )** no es suficiente.

Por consiguiente, tiene sentido complementar la clase **Arrays** con alguna utilidad adicional, que se ubicará por comodidad en el paquete **com.bruceeckel.util**. Estas utilidades permiten imprimir un array de cualquier tipo, y rellenan un array con valores u objetos creados por un objeto denominado *generador* que cada uno puede definir.

Dado que es necesario crear código para cada tipo primitivo al igual que para **Object**, hay muchísimo código prácticamente duplicado[3].Por ejemplo, se requiere una interfaz "generador" por cada tipo, puesto que el valor de retorno de **siguiente( )** debe ser distinto en cada caso:

```
//: com:bruceeckel:util:Generador.java
package com.bruceeckel.util;
public interface Generador {
  Object siguiente();
} ///:~

//: com:bruceeckel:util:GeneradorBoolean.java
package com.bruceeckel.util;
public interface GeneradorBoolean {
  boolean siguiente();
} ///:~

//: com:bruceeckel:util:GeneradorByte.java
package com.bruceeckel.util;
public interface GeneradorByte {
  byte siguiente();
} ///:~

//: com:bruceeckel:util:GeneradorChar.java
package com.bruceeckel.util;
public interface GeneradorChar {
  char siguiente();
} ///:~

//: com:bruceeckel:util:GeneradorShort.java
package com.bruceeckel.util;
public interface GeneradorShort {
  short siguiente();
} ///:~

//: com:bruceeckel:util:GeneradorInt.java
package com.bruceeckel.util;
public interface GeneradorInt {
  int siguiente();
} ///:~

//: com:bruceeckel:util:GeneradorLong.java
```

[3] El programador de C++ notará cuánto código podría colapsarse con la utilizacón de parámetros por defecto y plantillas. El programador de Python notará que esta biblioteca sería completamente innecesaria en este último lenguaje.

```
package com.bruceeckel.util;
public interface GeneradorLong {
  long siguiente();
} ///:~
```

```
//: com:bruceeckel:util:GeneradorFloat.java
package com.bruceeckel.util;
public interface GeneradorFloat {
  float siguiente();
} ///:~
```

```
//: com:bruceeckel:util:GeneradorDouble.java
package com.bruceeckel.util;
public interface GeneradorDouble {
  double siguiente();
} ///:~
```

Arrays2 contiene varias funciones **escribir( )**, sobrecargadas para cada tipo. Se puede simplemente imprimir un array, de forma que se pueda añadir un mensaje antes de que se imprima, o se puede imprimir un rango de elementos dentro de un array. El código de añadir método **escribir( )** es casi autoexplicatorio:

```
//: com:bruceeckel:util:Arrays2.java
// Un suplemento para java.util.Arrays, que proporciona
// funcionalidad adicional útil para trabajar
// con arrays. Permite imprimir un array,
// que puede ser rellenado a través un objeto "generador"
// definido por el usuario.
package com.bruceeckel.util;
import java.util.*;

public class Arrays2 {
  private static void
  start(int de, int para, int longitud) {
    if(de != 0 || para != longitud)
      System.out.print("["+ de +":"+ para +"] ");
    System.out.print("(");
  }
  private static void fin() {
    System.out.println(")");
  }
  public static void escribir(Object[] a) {
    escribir(a, 0, a.length);
  }
  public static void
  print(String mensaje, Object[] a) {
    System.out.escribir(mensaje + " ");
```

```
    escribir(a, 0, a.length);
  }
  public static void
  escribir(Object[] a, int de, int para){
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para -1)
        System.out.print(", ");
    }
    fin();
  }
  public static void escribir(boolean[] a) {
      escribir(a, 0, a.length);
  }
  public static void
  escribir(String mensaje, boolean[] a) {
    System.out.print(mensaje + " ");
    escribir(a, 0, a.length);
  }
  public static void
  escribir(boolean[] a, int de, int para) {
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para -1)
        System.out.print(", ");
    }
    fin();
  }
  public static void escribir(byte[] a) {
      escribir(a, 0, a.length);
  }
  public static void
  escribir(String mensaje, byte[] a) {
    System.out.print(mensaje + " ");
    escribir(a, 0, a.length);
  }
  public static void
  escribir(byte[] a, int de, int para) {
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para -1)
        System.out.print(", ");
```

```
    }
    fin();
  }
  public static void escribir(char[] a) {
      escribir(a, 0, a.length);
  }
  public static void
  escribir(String mensaje, char[] a) {
    System.out.print(mensaje + " ");
    escribir(a, 0, a.length);
  }
  public static void
  escribir(char[] a, int de, int para) {
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para -1)
        System.out.print(", ");
    }
    fin();
  }
  public static void escribir(short[] a) {
      escribir(a, 0, a.length);
  }
  public static void
  escribir(String mensaje, short[] a) {
    System.out.print(mensaje + " ");
    escribir(a, 0, a.length);
  }
  public static void
  escribir(short[] a, int de, int para) {
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para - 1)
        System.out.print(", ");
    }
    fin();
  }
  public static void escribir(int[] a) {
      escribir(a, 0, a.length);
  }
  public static void
  escribir(String mensaje, int[] a) {
    System.out.print(mensajes + " ");
```

```
    escribir(a, 0, a.length);
  }
  public static void
  escribir(int[] a, int de, int para) {
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para - 1)
        System.out.print(", ");
    }
    fin();
  }
  public static void escribir(long[] a) {
    escribir(a, 0, a.length);
  }
  public static void
  escribir(String mensaje, long[] a) {
    System.out.print(mensaje + " ");
    escribir(a, 0, a.length);
  }
  public static void
  escribir(long[] a, int de, int para) {
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para - 1)
        System.out.print(", ");
    }
    fin();
  }
  public static void escribir(float[] a) {
      escribir(a, 0, a.length);
  }
  public static void
  escribir(String mensaje, float[] a) {
    System.out.print(mensaje + " ");
    escribir(a, 0, a.length);
  }
  public static void
  escribir(float[] a, int de, int para) {
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para - 1)
        System.out.print(", ");
```

```
    }
    fin();
  }
  public static void escribir(double[] a) {
      escribir(a, 0, a.length);
  }
  public static void
  escribir(String mensaje, double[] a) {
    System.out.print(mensaje + " ");
    escribir(a, 0, a.length);
  }
  public static void
  escribir(double[] a, int de, int para){
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para - 1)
        System.out.print(", ");
    }
    fin();
  }
  // Rellenar un array utilizando un generador:
  public static void
  rellenar(Object[] a, Generador gen) {
      rellenar(a, 0, a.length, gen);
  }
  public static void
  rellenar(Object[] a, int de, int para,
        Generador gen){
    for(int i = de; i < para; i++)
      a[i] = gen.siguiente();
  }
  public static void
  rellenar(boolean[] a, GeneradorBoolean gen) {
      rellenar(a, 0, a.length, gen);
  }
  public static void
  rellenar(boolean[] a, int de, int para,
      GeneradorBoolean gen) {
    for(int i = de; i < para; i++)
      a[i] = gen.siguiente();
  }
  public static void
  rellenar(byte[] a, GeneradorByte gen) {
      rellenarl(a, 0, a.length, gen);
```

```
  }
  public static void
  rellenar(byte[] a, int de, int para,
      GeneradorByte gen) {
    for(int i = de; i < para; i++)
      a[i] = gen.siguiente();
  }
  public static void
  rellenar(char[] a, GeneradorChar gen) {
      rellenar(a, 0, a.length, gen);
  }
  public static void
  rellenar(char[] a, int de, int para,
      GeneradorChar gen) {
    for(int i = de; i < para; i++)
      a[i] = gen.siguiente();
  }
  public static void
  rellenar(short[] a, GeneradorShort gen) {
      rellenar(a, 0, a.length, gen);
  }
  public static void
  rellenar(short[] a, int de, int para,
      GeneradorShort gen) {
    for(int i = de; i < para; i++)
      a[i] = gen.siguiente();
  }
  public static void
  rellenar(int[] a, GeneradorInt gen) {
      rellenar(a, 0, a.length, gen);
  }
  public static void
  rellenar(int[] a, int de, int para,
      GeneradorInt gen) {
    for(int i = de; i < para; i++)
      a[i] = gen.siguiente();
  }
  public static void
  rellenar(long[] a, GeneratorLong gen) {
      rellenar(a, 0, a.length, gen);
  }
  public static void
  rellenar(long[] a, int de, int para,
      GeneradorLong gen) {
    for(int i = de; i < para; i++)
```

```
      a[i] = gen.siguiente();
  }
  public static void
  rellenar(float[] a, GeneradorFloat gen) {
      rellenar(a, 0, a.length, gen);
  }
  public static void
  rellenar(float[] a, int de, int para,
        GeneradorFloat gen) {
    for(int i = from; i < para; i++)
      a[i] = gen.siguiente();
  }
  public static void
  rellenar(double[] a, GeneradorDoble gen) {
      rellenar(a, 0, a.length, gen);
  }
  public static void
  rellenar(double[] a, int de, int para,
        DoubleGenerator gen){
    for(int i = de; i < para; i++)
      a[i] = gen.siguiente();
  }
  private static Random r = new Random();
  public static class GeneradorBooleanAleatorio
  implements GeneradorBoolean {
    public boolean siguiente() {
      return r.nextBoolean();
    }
  }
  public static class GeneradorByteAleatorio
  implements GeneradorByte {
    public byte siguiente() {
      return (byte)r.nextInt();
    }
  }
  static String fuente =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
    "abcdefghijklmnopqrstuvwxyz";
  static char[] fuenteArray = fuente.toCharArray();
  public static class GeneradorCharAleatorio
  implement GeneradorChar {
    public char siguiente() {
      int pos = Math.abs(r.nextInt());
      return fuenteArray[pos % fuenteArray.length];
    }
```

```
  }
  public static class GeneradorStringAleatorio
  implements Generador {
    private int lon;
    private GeneradorCharAleatorio cg =
      new GeneradorCharAleatorio();
    public GeneradorStringAleatorio(int longitud) {
      lon = longitud;
    }
    public Object siguiente() {
      char[] buf = new char[lon];
      for(int i = 0; i < lon; i++)
        buf[i] = cg.siguiente();
      return new String(buf);
    }
  }
  public static class GeneradorShorAleatorio
  implements GeneradorInt {
    public short siguiente() {
      return (short)r.nextInt();
    }
  }
  public static class GeneradorIntAleatorio
  implements GeneratorInt {
    private int mod = 10000;
    public GeneradorIntAleatorio() {}
    public GeneradorIntAleatorio(int modulo) {
      mod = modulo;
    }
    public int siguiente() {
      return r.nextInt() % mod;
    }
  }
  public static class GeneradorLongAleatorio
  implement GeneradorLong {
    public long siguiente() { return r.nextLong(); }
  }
  public static class GeneradorFloatAleatorio
  implements GeneradorFloat {
    public float siguiente() { return r.nextFloat(); }
  }
  public static class GeneradorDoubleAleatorio
  implements GeneradorDouble {
    public double siguiente() {return r.nextDouble();}
  }
} ///:~
```

Para rellenar un array utilizando un generador, el método **rellenar( )** toma una referencia a una **interfaz** generadora adecuada, que tiene un método **siguiente( )** que de alguna forma producirá un objeto del tipo correcto (dependiendo de cómo se implemente la interfaz). El método **rellenar( )** simplemente invoca a **siguiente( )** hasta que se ha rellenado el rango deseado. Ahora, se puede crear cualquier generador implementando la **interfaz** adecuada, y luego utilizar el generador con el método **rellenar( )**.

Los generadores de datos aleatorios son útiles para hacer pruebas, por lo que se crea un conjunto de clases internas para implementar las interfaces generadoras de datos primitivos, al igual que el generador de cadenas de caracteres **String** para representar a un **Objeto**. Se puede ver que **GeneradorStringAleatorio** usa **GeneradorCharAleatorio** para rellenar un array de caracteres, que después se convierte en una cadena de caracteres (**String**). El tamaño del array viene determinado por el parámetro pasado al constructor.

Para generar números que no sean demasiado grandes, **GeneradorIntAleatorio** toma por defecto un módulo de 10.000, pero el constructor sobrecargado permite seleccionar un valor menor.

He aquí un programa para probar la biblioteca y demostrar cómo se usa:

```
//: c09:PruebaArrays2.java
// Probar y demostrar las utilidades de Arrays2
import com.bruceeckel.util.*;

public class PruebaArrays2 {
  public static void main(String[] args) {
    int tamanio = 6;
    // O tomar el tamaño de la lista de parámetros:
    if(args.length != 0)
      tamanio= Integer.parseInt(args[0]);
    boolean[] a1 = new boolean[tamanio];
    byte[] a2 = new byte[tamanio];
    char[] a3 = new char[tamanio];
    short[] a4 = new short[tamanio];
    int[] a5 = new int[tamanio];
    long[] a6 = new long[tamanio];
    float[] a7 = new float[tamanio];
    double[] a8 = new double[tamanio];
    String[] a9 = new String[tamanio];
    Arrays2.rellenar(a1,
      new Arrays2.GeneradorBooleanAleatorio());
    Arrays2.escribir(a1);
    Arrays2.escribir("a1 = ", a1);
    Arrays2.escribir(a1, tamanio/3, tamanio/3 + tamanio/3);
    Arrays2.rellenar(a2,
      new Arrays2.GeneradorByteAleatorio());
    Arrays2.escribir(a2);
    Arrays2.escribir("a2 = ", a2);
    Arrays2.escribir(a2, tamanio/3, tamanio/3 + tamanio/3);
```

```
    Arrays2.rellenar(a3,
      new Arrays2.GeneradorCharAleatorio());
    Arrays2.escribir(a3);
    Arrays2.escribir("a3 = ", a3);
    Arrays2.escribir(a3, tamanio/3, tamanio/3 + tamanio/3);
    Arrays2.rellenar(a4,
      new Arrays2.GeneradorShortAleatorio());
    Arrays2.escribir(a4);
    Arrays2.escribir("a4 = ", a4);
    Arrays2.escribir(a4, tamanio/3, tamanio/3 + tamanio/3);
    Arrays2.rellenar(a5,
      new Arrays2.GeneradorIntAleatorio());
    Arrays2.escribir(a5);
    Arrays2.escribir("a5 = ", a5);
    Arrays2.escribir(a5, tamanio/3, tamanio/3 + tamanio/3);
    Arrays2.rellenar(a6,
      new Arrays2.GeneradorLongAleatorio());
    Arrays2.escribir(a6);
    Arrays2.escribir("a6 = ", a6);
    Arrays2.escribir(a6, tamanio/3,tamanio/3 + tamanio/3);
    Arrays2.rellenar(a7,
      new Arrays2.GeneradorFloatAleatorio());
    Arrays2.escribir(a7);
    Arrays2.escribir("a7 = ", a7);
    Arrays2.escribir(a7, tamanio/3, tamanio/3 + tamanio/3);
    Arrays2.rellenar(a8,
      new Arrays2.GeneradorDoubleAleatorio());
    Arrays2.escribir(a8);
    Arrays2.escribir("a8 = ", a8);
    Arrays2.escribir(a8, tamanio/3, tamanio/3 + tamanio/3);
    Arrays2.rellenar(a9,
      new Arrays2.GeneradorStringAleatorio(7));
    Arrays2.escribir(a9);
    Arrays2.escribir("a9 = ", a9);
    Arrays2.escribir(a9, tamanio/3, tamanio/3 + tamanio/3);
  }
} ///:~
```

El parámetro **tamanio** tiene un valor por defecto, pero también se puede establecer a partir de la línea de comandos.

Rellenar un array

La biblioteca estándar de Java **Arrays** también tiene un método **rellenar (fill( ))**, pero es bastante trivial —sólo duplica un único valor en cada posición, o en el caso de los objetos, copia la misma re-

ferencia a todas las posiciones. Utilizando **Arrays2.escribir( )**, se pueden demostrar fácilmente los métodos **Arrays.fill( )**:

```
//: c09:RellenarArrays.java
// Usando Arrays.fill()
import com.bruceeckel.util.*;
import java.util.*;

public class RellenarArrays {
  public static void main(String[] args) {
    int tamanio = 6;
    // O tomar el tamaño de la línea de comandos:
    if(args.length != 0)
      tamanio = Integer.parseInt(args[0]);
    boolean[] a1 = new boolean[tamanio];
    byte[] a2 = new byte[tamanio];
    char[] a3 = new char[tamanio];
    short[] a4 = new short[tamanio];
    int[] a5 = new int[tamanio];
    long[] a6 = new long[tamanio];
    float[] a7 = new float[tamanio];
    double[] a8 = new double[tamanio];
    String[] a9 = new String[tamanio];
    Arrays.fill(a1, true);
    Arrays2.escribir("a1 = ", a1);
    Arrays.fill(a2, (byte)11);
    Arrays2.escribir("a2 = ", a2);
    Arrays.fill(a3, 'x');
    Arrays2.escribir("a3 = ", a3);
    Arrays.fill(a4, (short)17);
    Arrays2.escribir("a4 = ", a4);
    Arrays.fill(a5, 19);
    Arrays2.escribir("a5 = ", a5);
    Arrays.fill(a6, 23);
    Arrays2.escribir("a6 = ", a6);
    Arrays.fill(a7, 29);
    Arrays2.escribir("a7 = ", a7);
    Arrays.fill(a8, 47);
    Arrays2.escribir("a8 = ", a8);
    Arrays.fill(a9, "Hola");
    Arrays2.escribir("a9 = ", a9);
    // Manipular rangos:
    Arrays.fill(a9, 3, 5, "Mundo");
    Arrays2.escribir("a9 = ", a9);
  }
} ///:~
```

Se puede o bien rellenar todo el array, o —como se ve en las dos últimas sentencias —un rango de elementos. Pero dado que sólo se puede proporcionar un valor a usar en el relleno si se usa **Arrays.fill( )**, los métodos **Arrays2.rellenar( )** producen resultados mucho más interesantes.

Copiar un array

La biblioteca estándar de Java proporciona un método **estático**, llamado **System.arraycopy( )**, que puede hacer copias mucho más rápidas de arrays que si se usa un bucle **for** para hacer la copia a mano. **System.arraycopy( )** está sobrecargado para manejar todos los tipos. He aquí un ejemplo que manipula un array de **enteros:**

```
//: c09:CopiarArrays.java
// Usando System.arraycopy()
import com.bruceeckel.util.*;
import java.util.*;

public class CopiarArrays {
  public static void main(String[] args) {
    int[] i = new int[25];
    int[] j = new int[25];
    Arrays.fill(i, 47);
    Arrays.fill(j, 99);
    Arrays2.escribir("i = ", i);
    Arrays2.escribir("j = ", j);
    System.arraycopy(i, 0, j, 0, i.length);
    Arrays2.escribir("j = ", j);
    int[] k = new int[10];
    Arrays.fill(k, 103);
    System.arraycopy(i, 0, k, 0, k.length);
    Arrays2.escribir("k = ", k);
    Arrays.fill(k, 103);
    System.arraycopy(k, 0, i, 0, k.length);
    Arrays2.print("i = ", i);
    // Objetos:
    Integer[] u = new Integer[10];
    Integer[] v = new Integer[5];
    Arrays.fill(u, new Integer(47));
    Arrays.fill(v, new Integer(99));
    Arrays2.escribir("u = ", u);
    Arrays2.escribir("v = ", v);
    System.arraycopy(v, 0,
      u, u.length/2, v.length);
    Arrays2.escribir("u = ", u);
  }
} ///:~
```

Los parámetros de **arraycopy( )** son el array fuente, el desplazamiento del array fuente a partir del cual comenzar la copia, el array de destino, el desplazamiento dentro del array de destino en el que comenzar a copiar, y el número de elementos a copiar. Naturalmente, cualquier violación de los límites del array causaría una excepción.

El ejemplo muestra que pueden copiarse, tanto los arrays de datos primitivos, como los de objetos. Sin embargo, si se copia un array de objetos, solamente se copian las referencias —no hay duplicación de los objetos en sí. A esto se le llama *copia superficial.* (Véase Apéndice A.)

Comparar arrays

La clase **Arrays** proporciona un método sobrecargado **equals( )** para comparar arrays enteros y ver si son iguales. Otra vez, se trata de un método sobrecargado para todos los tipos de datos primitivos y para **Objetos**. Para que dos arrays sean iguales, deben tener el mismo número de elementos y además cada elemento debe ser equivalente a su elemento correspondiente en el otro array, utilizando el método **equals( )** para cada elemento. (En el caso de datos primitivos, se usa la clase de su envoltorio **equals( )**; por ejemplo, se usa **Integer.equals( )** para **int**.) He aquí un ejemplo:

```
//: c09:CompararArrays.java
// Usando Arrays.equals()
import java.util.*;

public class CompararArrays {
  public static void main(String[] args) {
    int[] a1 = new int[10];
    int[] a2 = new int[10];
    Arrays.fill(a1, 47);
    Arrays.fill(a2, 47);
    System.out.println(Arrays.equals(a1, a2));
    a2[3] = 11;
    System.out.println(Arrays.equals(a1, a2));
    String[] s1 = new String[5];
    Arrays.fill(s1, "Hi");
    String[] s2 = {"Hi", "Hi", "Hi", "Hi", "Hi"};
    System.out.println(Arrays.equals(s1, s2));
  }
} ///:~
```

Originalmente, **a1** y **a2** son exactamente iguales, de forma que la salida es "verdadero", pero al cambiar uno de los elementos, la segunda línea de la salida será "falso". En este último caso, todos los elementos de **s1** apuntan al mismo objeto, pero **s2** tiene cinco únicos objetos. Sin embargo, la igualdad de los arrays se basa en los contenidos (mediante **Object.equals( )**) por lo que el resultado es "verdadero".

Comparaciones de elementos de arrays

Una de las características que faltan en las bibliotecas de Java 1.0 y 1.1 son las operaciones logarítmicas —incluso la ordenación simple. Ésta era una situación bastante confusa para alguien que esperara una biblioteca estándar adecuada. Afortunadamente, Java 2 remedia esta situación, al menos para el problema de ordenación.

El problema de escribir código de ordenación genérico consiste en que esa ordenación debe llevar a cabo comparaciones basadas en el tipo del objeto. Por supuesto, un enfoque es escribir un método de ordenación distinto para cada tipo distinto, pero habría que ser capaz de reconocer que esto no produce código fácilmente reutilizable para tipos nuevos.

Un primer objetivo del diseño de programación es "separar los elementos que cambian de los que permanecen igual", y aquí el código que sigue igual es el algoritmo general de ordenación, pero lo que cambia de un uso al siguiente es la forma de comparar los objetos. Por tanto, en vez de incluir el código de comparación en muchas rutinas de ordenación se usa la técnica de *retrollamadas*. Con una llamada hacia atrás, la parte de código que varía de caso a caso está encapsulada dentro de su propia clase, y la parte de código que es siempre la misma puede invocar hacia atrás al código que cambia. De esa forma, se pueden hacer objetos diferentes para expresar distintas formas de comparación y alimentar con ellos el mismo código de ordenación.

En Java 2 hay dos formas de proporcionar funcionalidad de comparación. La primera es con el *método de comparación natural*, que se comunica con una clase implementando la interfaz **java.lang.Comparable**. Se trata de una interfaz bastante simple con un único método **compareTo( )**. Este método toma otro **Objeto** como parámetro, y produce un valor negativo si el parámetro es menor que el objeto actual, cero si el parámetro es igual, y un valor positivo si el parámetro es mayor que el objeto actual.

He aquí una clase que implementa **Comparable** y demuestra la comparación utilizando el método **Arrays.sort( )** de la biblioteca estándar de Java:

```
//: c09:TipoComp.java
// Implementando Comparable en una clase.
import com.bruceeckel.util.*;
import java.util.*;

public class TipoComp implements Comparable {
  int i;
  int j;
  public TipoComp(int n1, int n2) {
    i = n1;
    j = n2;
  }
  public String toString() {
    return "[i = " + i + ", j = " + j + "]";
  }
```

```
  public int compareTo(Object rv) {
    int rvi = ((TipoComp)rv).i;
    return (i < rvi ? -1 : (i == rvi ? 0 : 1));
  }
  private static Random r = new Random();
  private static int intAleatorio() {
    return Math.abs(r.nextInt()) % 100;
  }
  public static Generador generador() {
    return new Generador() {
      public Object siguiente() {
        return new TipoComp(intAleatorio(),intAleatorio());
      }
    };
  }
  public static void main(String[] args) {
    TipoComp[] a = new TipoComp[10];
    Arrays2.rellenar(a, generador());
    Arrays2.escribir("antes de ordenar, a = ", a);
    Arrays.sort(a);
    Arrays2.escribir("despues de ordenar, a = ", a);
  }
} ///:~
```

Cuando se define la función de comparación, uno es responsable de decidir qué significa comparar un objeto con otro. Aquí, sólo se usan los valores **i** para la comparación, ignorando los valores **j**.

El método **estático intAleatorio( )** produce valores positivos entre 0 y 100, y el método **generador( )** produce un objeto que implementa la interfaz **Generador**, creando una clase interna anónima (ver Capítulo 8). Así se crean objetos **TipoComp** inicializándolos con valores al azar. En **main ( )** se usa el generador para rellenar un array de **TipoComp**, que se ordena después. Si no se hubiera implementado **Comparable**, se habría obtenido un mensaje de error de tiempo de compilación al tratar de invocar a **sort( )**.

Ahora, supóngase que alguien nos pasa una clase que no implementa **Comparable**, o nos pasan esta clase que *sí* que implementa **Comparable**, pero decide que no le gusta cómo funciona y preferiríamos una función de comparación distinta. Para lograrlo, se usa el segundo enfoque de comparación de objetos, creando una clase separada que implementa una interfaz denominada **Comparator**. Ésta tiene dos métodos, **compare( )** y **equals( )**. Sin embargo, no se tiene que implementar **equals( )** excepto por necesidades de rendimiento especiales, dado que cada vez que se crea la clase ésta se hereda implícitamente de **Object**, que tiene un **equals( )**. Por tanto, se puede usar el método por defecto **equals( )** que devuelve un **object** y satisfacer el contrato impuesto por la interfaz.

La clase **Collections** (que se estudiará más tarde) contiene un **Comparator** simple que invierte el orden de ordenación natural. Éste se puede aplicar de manera sencilla al **TipoComp**:

```
//: c09:Inverso.java
// El comparador Collecions.reverseOrder().
import com.bruceeckel.util.*;
import java.util.*;

public class Inverso {
  public static void main(String[] args) {
    TipoComp[] a = new TipoComp[10];
    Arrays2.rellenar(a, TipoComp.generador());
    Arrays2.escribir("antes de ordenar, a = ", a);
    Arrays.sort(a, Collections.reverseOrder());
    Arrays2.escribir("despues de ordenar, a = ", a);
  }
} ///:~
```

La llamada a **Collections.reverseOrder( )** produce la referencia al **Comparator**.

Como segundo ejemplo, el **Comparator** siguiente compara objetos **TipoComp** basados en sus valores **j** en vez de en sus valores **i**:

```
//: c09:PruebaComparador.java
// Implementando un Comparator para una clase.
import com.bruceeckel.util.*;
import java.util.*;

class ComprobadorTipoComp implements Comparator {
  public int compare(Object o1, Object o2) {
    int j1 = ((TipoComp)o1).j;
    int j2 = ((TipoComp)o2).j;
    return (j1 < j2 ? -1 : (j1 == j2 ? 0 : 1));
  }
}

public class PruebaComparador {
  public static void main(String[] args) {
    TipoComp[] a = new TipoComp[10];
    Arrays2.rellenar(a, TipoComp.generador());
    Arrays2.escribir("antes de ordenar, a = ", a);
    Arrays.sort(a, new ComparadorTipoComp());
    Arrays2.escribir("despues de ordenar, a = ", a);
  }
} ///:~
```

El método **compare( )** debe devolver un entero negativo, cero o un entero positivo si el primer parámetro es menor que, igual o mayor que el segundo, respectivamente.

Ordenar un array

Con los métodos de ordenación incluidos, se puede ordenar cualquier array de tipos primitivos, y un array de objetos que, o bien implemente **Comparable,** o bien tenga un **Comparator** asociado. Éste rellena un gran agujero en las bibliotecas Java —se crea o no, ¡en Java 1.0 y 1.1 no había soporte para ordenar cadenas de caracteres! He aquí un ejemplo que genera objetos **String** y los ordena:

```
//: c09:OrdenarStrings.java
// Ordenando un array de Strings.
import com.bruceeckel.util.*;
import java.util.*;

public class OrdenarStrings {
  public static void main(String[] args) {
    String[] sa = new String[30];
    Arrays2.rellenar(sa,
      new Arrays2.GeneradorStringAleatorio(5));
    Arrays2.escribir("Antes de ordenar: ", sa);
    Arrays.sort(sa);
    Arrays2.escribir("Despues de ordenar: ", sa);
  }
} ///:~
```

Algo que uno notará de la salida del algoritmo de ordenación de cadenas de caracteres es que es *lexicográfico*, por lo que coloca en primer lugar las palabras que empiezan con letras mayúsculas, seguidas de todas las palabras que empiezan con minúsculas. (Las guías telefónicas suelen ordenarse así.) También se podría desear agrupar todas las palabras juntas independientemente de si empiezan con mayúsculas o minúsculas, lo cual se puede hacer definiendo una clase **Comparator**, y por consiguiente, sobrecargando el comportamiento por defecto de **Comparable** para cadenas de caracteres. Para su reutilización, ésta se añadirá al paquete "util":

```
//: com:bruceeckel:util:ComparadorAlfabetico.java
// Manteniendo juntas las letras mayúsculas y minúsculas.
package com.bruceeckel.util;
import java.util.*;

public class ComparadorAlfabetico
implements Comparador{
  public int compare(Object o1, Object o2) {
    String s1 = (String)o1;
    String s2 = (String)o2;
    return s1.toLowerCase().compareTo(
      s2.toLowerCase());
  }
} ///:~
```

Cada **String** se convierte a minúsculas antes de esta comparación. El método **compareTo( )** incluido en **String** proporciona la funcionalidad deseada.

He aquí una prueba usando **ComparadorAlfabetico:**

```
//: c09:OrdenaAlfabeticamente.java
// Mantiene juntas las letras mayúsculas y minúsculas
import com.bruceeckel.util.*;
import java.util.*;

public class OrdenaAlfabeticamente {
  public static void main(String[] args) {
    String[] sa = new String[30];
    Arrays2.rellenar(sa,
      new Arrays2.GeneradorStringAleatorio(5));
    Arrays2.escribir("Antes de ordenar: ", sa);
    Arrays.sort(sa, new ComparadorAlfabetico());
    Arrays2.escribir("Despues de ordenar: ", sa);
  }
} ///:~
```

El algoritmo de ordenación usado en la biblioteca estándar de Java se diseñó para ser óptimo para el tipo de datos en particular a ordenar —algoritmo de ordenación rápida (Quicksort) para tipos primitivos, y un método de ordenación por mezcla *(merge sort)* en el caso de objetos. Por tanto, no sería necesario preocuparse por el rendimiento a menos que alguna herramienta de optimización indicara que el proceso de ordenación constituye un cuello de botella.

Buscar en un array ordenado

Una vez ordenado el array, se puede llevar a cabo una búsqueda rápida de algún elemento dentro del mismo utilizando **Arrays.binarySearch( )**. Sin embargo, es muy importante que no se trate de hacer uso de **binarySearch( )** en un array sin ordenar; los resultados serían impredecibles. El ejemplo siguiente usa un **GeneradorIntAleatorio** para rellenar un array, y después produce valores a buscar en el mismo:

```
//: c09:BuscarEnArray.java
// Usando Arrays.binarySearch().
import com.bruceeckel.util.*;
import java.util.*;

public class BuscarEnArray {
  public static void main(String[] args) {
    int[] a = new int[100];
    Arrays2. GeneradorIntAleatorio gen =
      new Arrays2.GeneradorIntAleatorio (1000);
    Arrays2.rellenar(a, gen);
```

```
      Arrays.sort(a);
      Arrays2.escribir("Array ordenado: ", a);
      while(true) {
        int r = gen.siguiente();
        int posicion = Arrays.binarySearch(a, r);
        if(posicion >= 0) {
          System.out.println("Localizacion de " + r +
            " es " + posicion + ", a[" +
            posicion + "] = " + a[posicion]);
          break; // sale del bucle while
        }
      }
    }
} ///:~
```

En el bucle **while** se generan valores aleatorios como datos a buscar, hasta que se encuentre uno de ellos.

Arrays.binarySearch( ) produce un valor mayor o igual a cero si se encuentra el elemento. Sino, produce un valor negativo que representa el lugar en el que debería insertarse el elemento si se estuviera manteniendo el array ordenado a mano. El valor producido es:

```
-(punto de inserción) - 1
```

El punto de inserción es el índice del primer elemento mayor que la clave, o **a.size( )**, si todos los elementos del array son menores que la clave especificada.

Si el array contiene elementos duplicados, no hay garantía de cuál será el que se localice. El algoritmo, por tanto, no está realmente diseñado para soportar elementos duplicados, aunque los tolera. Sin embargo, si se necesita una lista ordenada de elementos no duplicados, hay que usar un **TreeSet**, que se presentará más adelante en este capítulo. Éste se encarga de todos los detalles por ti automáticamente. El **TreeSet** sólo deberá ser reemplazado por un array mantenido a mano en aquellos casos en que haya cuellos de botella relacionados con el rendimiento.

Si se ha ordenado un array utilizando un **Comparador** (los arrays de tipos primitivos no permiten ordenaciones con un **Comparador**), hay que incluir el mismo **Comparador** al hacer una **binarySearch( )** (utilizando la versión sobrecargada que se proporciona de la función). Por ejemplo, el programa **OrdenarAlfabeticamente.java** puede cambiarse para que lleve a cabo una búsqueda:

```
//: c09:BuscarAlfabeticamente.java
// Buscar con un comparador.
import com.bruceeckel.util.*;
import java.util.*;

public class BuscarAlfabeticamente {
  public static void main(String[] args) {
```

```
    String[] sa = new String[30];
    Arrays2.rellenar(sa,
      new Arrays2.GeneradorStringAleatorio(5));
    ComparadorAlfabetico comp =
      new ComparadorAlfabetico ();
    Arrays.sort(sa, comp);
    int indice =
      Arrays.binarySearch(sa, sa[10], comp);
    System.out.println("Indice = " + indice);
  }
} ///:~
```

Debe pasarse el **Comparador** al **binarySearch( )** como tercer parámetro. En el ejemplo de arriba, se garantiza el éxito porque el elemento de búsqueda se ha arrancado del propio array.

Resumen de arrays

Para resumir lo visto hasta el momento, la primera y más eficiente selección a la hora de mantener un grupo de objetos debería ser un array, e incluso uno se ve obligado a seguir esta elección si lo que se desea guardar es un conjunto de datos primitivos. En el resto de este capítulo, se echará un vistazo al caso más general, en el que no se sabe en el momento de escribir el programa cuántos objetos se necesitarán o si se necesitará una forma más sofisticada de almacenamiento de los objetos. Java proporciona una biblioteca de *clases contenedoras* para solucionar este problema, en la que destacan los tipos básicos **List**, **Set** y **Map**. Utilizando estas herramientas se puede solucionar una cantidad de problemas sorprendente.

Entre sus otras características —**Set**, por ejemplo, sólo guarda un objeto de cada valor, y **Map** es un *array asociativo* que permite asociar cualquier objeto con cualquier otro— las clases contenedoras de Java redefinirán su tamaño automáticamente. Por tanto, y a diferencia de los arrays, se puede meter cualquier número de objetos y no hay que preocuparse del tamaño del contenedor cuando se está escribiendo el programa.

Introducción a los contenedores

Las clases contenedoras son una de las herramientas más potentes de cara al desarrollo puro y duro, puesto que incrementan significativamente el potencial programador. Los contenedores de Java 2 representan un rediseño[4] concienzudo de las pobres muestras de Java 1.0 y 1.1. Algunos de los rediseños han provocado mayores restricciones y precisan de un mayor cuidado. También completa la funcionalidad de la biblioteca de contenedores, proporcionando el comportamiento de las listas enlazadas y colas (con doble extremo llamadas "bicolas").

[4] Realizado Joshua Bloch en Sun.

El diseño de una biblioteca de contenedores es complicado (al igual que ocurre en la mayoría de problemas de diseño de bibliotecas). En C++, las clases contenedoras cubrían las bases con muchas clases distintas. Esto era mejor que lo que estaba disponible antes de las clases contenedoras de C++ (nada), pero no se traducía bien a Java. Por otro lado, he visto una biblioteca de contenedores consistente en una única clase, "contenedor", que actúa tanto de secuencia lineal, como de array asociativo simultáneamente. La biblioteca contenedora de Java 2 mejora el balance: se obtiene la funcionalidad completa deseada para una biblioteca de contenedores madura, y es más fácil de aprender y utilizar que las bibliotecas de clases contenedoras, y otras bibliotecas de contenedores semejantes. En ocasiones el resultado podría parecer algo extraño. A diferencia de otras decisiones hechas para las primeras bibliotecas de Java, estas extrañezas no eran accidentes, sino decisiones cuidadosamente consideradas basadas en compromisos de complejidad. Podría llevar algún tiempo adaptarse a algunos aspectos de la biblioteca, pero pienso que pronto nos haremos al uso de estas nuevas herramientas.

La biblioteca contenedora de Java 2 toma la labor de "almacenar objetos" y lo divide en dos conceptos distintos:

1. Colección **(Collection)**: grupo de elementos individuales, a los que generalmente se aplica alguna regla. Una lista **(List)** debe contener elementos en una secuencia concreta, y un conjunto **(Set)** no puede tener elementos duplicados. (Una *bolsa*, que no está implementada en la biblioteca de contenedores de Java —puesto que las **Listas** proporcionan gran parte de esta funcionalidad— no tiene estas reglas.)

2. Mapa **(Map)**: grupo de pares de objetos clave-valor. A primera vista, esto parecería una Colección **(Collection)** de pares, pero cuando se intenta implementar así, su diseño se vuelve complicado, por lo que es mejor convertirla en un concepto separado. Por otro lado, es conveniente buscar porciones de **Mapa** creando una **Colección** que la represente. Por consiguiente, un **Mapa** puede devolver un Conjunto **(Set)** de sus claves, una **Colección** de sus valores, o un **Conjunto** de sus pares. Los **Mapa**s, al igual que los arrays pueden extenderse de manera sencilla a múltiples dimensiones sin añadir nuevos conceptos: simplemente se construye un **Mapa** cuyos valores son **Mapa**s (y el valor de *esos* **Mapa**s pueden ser **Mapa**s, etc.).

Primero se echará un vistazo a las caracteríticas generales de los contenedores, después se presentarán los detalles, y finalmente se aprenderá por qué hay distintas versiones de algunos contenedores, y cómo elegir entre las mismas.

Visualizar contenedores

A diferencia de los arrays, los contenedores se visualizan elegantemente sin necesidad de ayuda. He aquí un ejemplo que muestra también los tipos básicos de contenedores:

```
//: c09:ImprimirContenedores.java
// Los contenedores se imprimen a sí mismos automáticamente.
import java.util.*;

public class ImprimirContenedores {
  static Collection rellenar(Collection c) {
```

```
    c.add("perro");
    c.add("perro");
    c.add("gato");
    return c;
  }
  static Map rellenar(Map m) {
    m.put("perro", "Bosco");
    m.put("perro", "Spot");
    m.put("gato", "Rags");
    return m;
  }
  public static void main(String[] args) {
    System.out.println(rellenar(new ArrayList()));
    System.out.println(rellenar(new HashSet()));
    System.out.println(rellenar(new HashMap()));
  }
} ///:~
```

Como se mencionó anteriormente, hay dos categorías básicas en la biblioteca de contenedores de Java. La distinción se basa en el número de elementos que se mantienen en cada posición del contenedor. La categoría **Colección** sólo mantiene un elemento en cada posición (el nombre es un poco lioso dado que a las propias bibliotecas de contenedores se les suele llamar también "colecciones"). Esta categoría incluye la **Lista**, que guarda un conjunto de elementos en una secuencia específica, y el **Conjunto**, que sólo permite la inserción de un elemento de cada tipo. La lista de Arrays (ArrayList) es un tipo de **Lista**, y el conjunto **Hash HashSet** es un tipo de **Conjunto**. Para añadir elementos a cualquier **Colección**, hay un método **add( )**.

El **Mapa** guarda pares de valores clave, de manera análoga a una mini base de datos. El programa de arriba usa una versión de **Mapa**, el **HashMap** Si se tiene un **Mapa** que asocia estados con sus capitales y se desea conocer la capital de Ohio, se mira en él —casi como si se estuviera haciendo un acceso indexado a un array. (A los **Mapa**s también se les denomina *arrays asociativos*.) Para añadir elementos a un **Mapa** hay un método **put( )** que toma una clave y un valor como argumentos. El ejemplo de arriba sólo muestra la inserción de elementos y no busca los elementos una vez añadidos éstos. Eso se mostrará más adelante.

Los métodos sobrecargados **fill( )** rellenan **Colecciones** y **Mapa**s respectivamente. Si se mira a la salida puede verse que el comportamiento impresor por defecto (proporcionado a través de los varios métodos **toString( )** de los contenedores) produce resultados bastante legibles, por lo que no es necesario un soporte adicional de impresión, como ocurre con los arrays:

```
[perro, perro, gato]
[gato, perro]
{gato=Rags, perro=Spot}
```

Una **Colección** siempre se imprime entre corchetes, separando cada elemento por comas. Un **Mapa** se imprime entre llaves, con cada clave y valor asociados mediante un signo igual (claves a la izquierda, valores a la derecha).

Se puede ver inmediatamente el comportamiento básico de cada contenedor. La **Lista** guarda los objetos exactamente tal y como se introducen, sin reordenamientos o ediciones. El **Conjunto**, sin embargo, sólo acepta uno de cada objeto y usa su propio método de ordenación interno (en general, a uno sólo le importa si algo es miembro o no del **Conjunto**, y no el orden en que aparece —para lo que se usaría una **Lista**). Y el **Mapa** sólo acepta un elemento de cada tipo, basándose en la clave, y tiene también su propia ordenación interna y no le importa el orden en que se introduzcan los elementos.

Rellenar contenedores

Aunque ya se ha resuelto el problema de impresión de los contenedores, el problema del rellenado de los mismos sufre de la misma deficiencia que **java.util.Arrays**. Exactamente igual que ocurre con **Arrays**, hay una clase denominada **Collections** que contiene métodos de utilidad **estaticos** incluyendo uno denominado **fill( )**. Este **fill( )** también se limita a duplicar una única referencia a un objeto a través de todo el contenedor, y funciona para objetos **Lista**, y no para **Conjuntos** o **Mapas**;

```
//: c09:RellenarListas.java
// El método Collections.fill().
import java.util.*;

public class RellenarListas{
  public static void main(String[] args) {
    Lista lista = new ArrayList();
    for(int i = 0; i < 10; i++)
      lista.add("");
    Collections.fill(lista, "Hola");
    System.out.println(lista);
  }
} ///:~
```

Este método es incluso menos útil de lo ya visto, debido al hecho de que sólo puede reemplazar elementos que ya se encuentran en la **Lista**, y no añadirá elementos nuevos.

Para crear ejemplos interesantes, he aquí una biblioteca complementaria **Colecciones2** (que es a su vez parte de **com.bruceeckel.util** por conveniencia) con un método **rellenar( ) (fill( ))**que usa un generador para añadir elementos, y permite especificar el número de elementos que se desea añadir. La **interfaz Generador** definida previamente, funcionará para **Colecciones**, pero el **Mapa** requiere su propia **interfaz** generadora puesto que hay que producir un par de objetos (una clave y un valor) por cada llamada a **siguiente( )**. He aquí la clase **Par**:

```
//: com:bruceeckel:util:Par.java
package com.bruceeckel.util;
public class Par {
  public Object clave, valor;
  Par(Object k, Object v) {
    clave = k;
    valor = v;
```

```
  }
} ///:~
```

A continuación, la **interfaz** generadora que produce el **Par**:

```
//: com:bruceeckel:util:GeneradorMapa.java
package com.bruceeckel.util;
public interface GeneradorMapa {
  Par siguiente();
} ///:~
```

Con estas clases, se puede desarrollar un conjunto de utilidades para trabajar con las clases contenedoras:

```
//: com:bruceeckel:util:Colecciones2.java
// Para rellenar cualquier tipo de contenedor
// usando un objeto generador.
package com.bruceeckel.util;
import java.util.*;

public class Colecciones2 {
  // Rellenar un array usando un generador:
  public static void
  rellenar(Collection c, Generator gen, int cont) {
    for(int i = 0; i < cont; i++)
      c.add(gen.next());
  }
  public static void
  rellenar(Map m, GeneradorMapa gen, int cont) {
    for(int i = 0; i < cont; i++) {
      Par p = gen.siguiente();
      m.put(p.clave, p.valor);
    }
  }
  public static class GeneradorParStringAleatorio
  implements MapGenerator {
    private Arrays2.GeneradorParStringAleatorio gen;
    public GeneradorParStringAleatorio(int lon) {
      gen = new Arrays2.GeneradorStringAleatorio(len);
    }
    public Par siguiente() {
      return new Par(gen.next(), gen.next());
    }
  }
  // Objeto por defecto con lo que no hay que
  // crear uno propio:
```

```
  public static GeneradorParStringAleatorio rsp =
    new GeneradorParStringAleatorio(10);
  public static class StringPairGenerator
  implements GeneradorMapa {
    private int indice = -1;
    private String[][] d;
    public GeneradorParString(String[][] datos) {
      d = datos;
    }
    public Par siguiente() {
      // Forzar que el índice sea envolvente:
      indice = (indice + 1) % d.length;
      return new Par(d[indice][0], d[indice][1]);
    }
    public GeneradorParString inicializar() {
      indice = -1;
      return this;
    }
  }
  // Usar un conjunto de datos predefinido:
  public static GeneradorParString geografia =
    new GeneradorParString(
      CapitalesPaises.pares);
  // Producir una secuencia a partir de un array 2D:
  public static class GeneradorString
  implements Generator {
    private String[][] d;
    private int posicion;
    private int indice = -1;
    public
    GeneradorString(String[][] datos, int pos) {
      d = datos;
      posicion = pos;
    }
    public Object siguiente() {
      // Forzar que el índice sea envolvente:
      indice = (indice + 1) % d.length;
      return d[indice][posicion];
    }
    public GeneradorString inicializar() {
      indice = -1;
      return this;
    }
  }
```

```
  // Usar un conjunto de datos predefinido:
  public static GeneradorString paises =
    new GeneradorString(CapitalesPaises.pares,0);
  public static GeneradorString capitales =
    new GeneradorString(CapitalesPaises.pares,1);
} ///:~
```

Ambas versiones de **rellenar( )** toman un argumento que determina el número de *datos* a añadir al contenedor. Además, hay dos generadores para el mapa: **GeneradorParStringAleatorio**, que crea cualquier número de pares de **cadenas de caracteres** galimatías de longitud determinada por el parámetro del constructor, y **GeneradorParString**, que produce pares de **cadenas de caracteres** a partir de un array bidimensional de **String**. El **GeneradorString** también toma un array bidimensional de **cadenas de caracteres** pero genera datos simples en vez de **Pares**. Los objetos **estáticos geografía**, **países** y **capitales** proporcionan generadores preconstruidos, los últimos tres utilizando todos los países del mundo y sus capitales. Fíjese que si se intentan crear más pares de los disponibles, los generadores volverán al comienzo, y si se están introduciendo pares en **Mapa**, simplemente se ignorarían los duplicados.

He aquí el conjunto de datos predefinido, que consiste en nombres de países y sus capitales. Está escrito con fuentes de tamaño pequeño para evitar que ocupe demasiado espacio:

```
//: com:bruceeckel:util:CapitalesPaises.java
package com.bruceeckel.util;
public class CapitalesPaises {
  public static final String[][] pares = {
    // Africa
    {"ALGERIA","Algiers"}, {"ANGOLA","Luanda"},
    {"BENIN","Porto-Novo"}, {"BOTSWANA","Gaberone"},
    {"BURKINA FASO","Ouagadougou"}, {"BURUNDI","Bujumbura"},
    {"CAMEROON","Yaounde"}, {"CAPE VERDE","Praia"},
    {"CENTRAL AFRICAN REPUBLIC","Bangui"},
    {"CHAD","N'djamena"}, {"COMOROS","Moroni"},
    {"CONGO","Brazzaville"}, {"DJIBOUTI","Dijibouti"},
    {"EGYPT","Cairo"}, {"EQUATORIAL GUINEA","Malabo"},
    {"ERITREA","Asmara"}, {"ETHIOPIA","Addis Ababa"},
    {"GABON","Libreville"}, {"THE GAMBIA","Banjul"},
    {"GHANA","Accra"}, {"GUINEA","Conakry"},
    {"GUINEA","~"}, {"BISSAU","Bissau"},
    {"CETE D'IVOIR (IVORY COAST)","Yamoussoukro"},
    {"KENYA","Nairobi"}, {"LESOTHO","Maseru"},
    {"LIBERIA","Monrovia"}, {"LIBYA","Tripoli"},
    {"MADAGASCAR","Antananarivo"}, {"MALAWI","Lilongwe"},
    {"MALI","Bamako"}, {"MAURITANIA","Nouakchott"},
    {"MAURITIUS","Port Louis"}, {"MOROCCO","Rabat"},
    {"MOZAMBIQUE","Maputo"}, {"NAMIBIA","Windhoek"},
    {"NIGER","Niamey"}, {"NIGERIA","Abuja"},
```

```
{"RWANDA","Kigali"}, {"SAO TOME E PRINCIPE","Sao Tome"},
{"SENEGAL","Dakar"}, {"SEYCHELLES","Victoria"},
{"SIERRA LEONE","Freetown"}, {"SOMALIA","Mogadishu"},
{"SOUTH AFRICA","Pretoria/Cape Town"}, {"SUDAN","Khartoum"},
{"SWAZILAND","Mbabane"}, {"TANZANIA","Dodoma"},
{"TOGO","Lome"}, {"TUNISIA","Tunis"},
{"UGANDA","Kampala"},
{"DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)","Kinshasa"},
{"ZAMBIA","Lusaka"}, {"ZIMBABWE","Harare"},
// Asia
{"AFGHANISTAN","Kabul"}, {"BAHRAIN","Manama"},
{"BANGLADESH","Dhaka"}, {"BHUTAN","Thimphu"},
{"BRUNEI","Bandar Seri Begawan"}, {"CAMBODIA","Phnom Penh"},
{"CHINA","Beijing"}, {"CYPRUS","Nicosia"},
{"INDIA","New Delhi"}, {"INDONESIA","Jakarta"},
{"IRAN","Tehran"}, {"IRAQ","Baghdad"},
{"ISRAEL","Jerusalem"}, {"JAPAN","Tokyo"},
{"JORDAN","Amman"}, {"KUWAIT","Kuwait City"},
{"LAOS","Vientiane"}, {"LEBANON","Beirut"},
{"MALAYSIA","Kuala Lumpur"}, {"THE MALDIVES","Male"},
{"MONGOLIA","Ulan Bator"}, {"MYANMAR (BURMA)","Rangoon"},
{"NEPAL","Katmandu"}, {"NORTH KOREA","P'yongyang"},
{"OMAN","Muscat"}, {"PAKISTAN","Islamabad"},
{"PHILIPPINES","Manila"}, {"QATAR","Doha"},
{"SAUDI ARABIA","Riyadh"}, {"SINGAPORE","Singapore"},
{"SOUTH KOREA","Seoul"}, {"SRI LANKA","Colombo"},
{"SYRIA","Damascus"}, {"TAIWAN (REPUBLIC OF CHINA)","Taipei"},
{"THAILAND","Bangkok"}, {"TURKEY","Ankara"},
{"UNITED ARAB EMIRATES","Abu Dhabi"}, {"VIETNAM","Hanoi"},
{"YEMEN","Sana'a"},
// Australia and Oceania
{"AUSTRALIA","Canberra"}, {"FIJI","Suva"},
{"KIRIBATI","Bairiki"},
{"MARSHALL ISLANDS","Dalap-Uliga-Darrit"},
{"MICRONESIA","Palikir"}, {"NAURU","Yaren"},
{"NEW ZEALAND","Wellington"}, {"PALAU","Koror"},
{"PAPUA NEW GUINEA","Port Moresby"},
{"SOLOMON ISLANDS","Honaira"}, {"TONGA","Nuku'alofa"},
{"TUVALU","Fongafale"}, {"VANUATU","< Port-Vila"},
{"WESTERN SAMOA","Apia"},
// Eastern Europe and former USSR
{"ARMENIA","Yerevan"}, {"AZERBAIJAN","Baku"},
{"BELARUS (BYELORUSSIA)","Minsk"}, {"GEORGIA","Tbilisi"},
{"KAZAKSTAN","Almaty"}, {"KYRGYZSTAN","Alma-Ata"},
{"MOLDOVA","Chisinau"}, {"RUSSIA","Moscow"},
```

```
{"TAJIKISTAN","Dushanbe"}, {"TURKMENISTAN","Ashkabad"},
{"UKRAINE","Kyiv"}, {"UZBEKISTAN","Tashkent"},
// Europe
{"ALBANIA","Tirana"}, {"ANDORRA","Andorra la Vella"},
{"AUSTRIA","Vienna"}, {"BELGIUM","Brussels"},
{"BOSNIA","-"}, {"HERZEGOVINA","Sarajevo"},
{"CROATIA","Zagreb"}, {"CZECH REPUBLIC","Prague"},
{"DENMARK","Copenhagen"}, {"ESTONIA","Tallinn"},
{"FINLAND","Helsinki"}, {"FRANCE","Paris"},
{"GERMANY","Berlin"}, {"GREECE","Athens"},
{"HUNGARY","Budapest"}, {"ICELAND","Reykjavik"},
{"IRELAND","Dublin"}, {"ITALY","Rome"},
{"LATVIA","Riga"}, {"LIECHTENSTEIN","Vaduz"},
{"LITHUANIA","Vilnius"}, {"LUXEMBOURG","Luxembourg"},
{"MACEDONIA","Skopje"}, {"MALTA","Valletta"},
{"MONACO","Monaco"}, {"MONTENEGRO","Podgorica"},
{"THE NETHERLANDS","Amsterdam"}, {"NORWAY","Oslo"},
{"POLAND","Warsaw"}, {"PORTUGAL","Lisbon"},
{"ROMANIA","Bucharest"}, {"SAN MARINO","San Marino"},
{"SERBIA","Belgrade"}, {"SLOVAKIA","Bratislava"},
{"SLOVENIA","Ljujiana"}, {"SPAIN","Madrid"},
{"SWEDEN","Stockholm"}, {"SWITZERLAND","Berne"},
{"UNITED KINGDOM","London"}, {"VATICAN CITY","---"},
// North and Central America
{"ANTIGUA AND BARBUDA","Saint John's"}, {"BAHAMAS","Nassau"},
{"BARBADOS","Bridgetown"}, {"BELIZE","Belmopan"},
{"CANADA","Ottawa"}, {"COSTA RICA","San Jose"},
{"CUBA","Havana"}, {"DOMINICA","Roseau"},
{"DOMINICAN REPUBLIC","Santo Domingo"},
{"EL SALVADOR","San Salvador"}, {"GRENADA","Saint George's"},
{"GUATEMALA","Guatemala City"}, {"HAITI","Port-au-Prince"},
{"HONDURAS","Tegucigalpa"}, {"JAMAICA","Kingston"},
{"MEXICO","Mexico City"}, {"NICARAGUA","Managua"},
{"PANAMA","Panama City"}, {"ST. KITTS","-"},
{"NEVIS","Basseterre"}, {"ST. LUCIA","Castries"},
{"ST. VINCENT AND THE GRENADINES","Kingstown"},
{"UNITED STATES OF AMERICA","Washington, D.C."},
// South America
{"ARGENTINA","Buenos Aires"},
{"BOLIVIA","Sucre (legal)/La Paz(administrative)"},
{"BRAZIL","Brasilia"}, {"CHILE","Santiago"},
{"COLOMBIA","Bogota"}, {"ECUADOR","Quito"},
```

```
    {"GUYANA","Georgetown"}, {"PARAGUAY","Asuncion"},
    {"PERU","Lima"}, {"SURINAME","Paramaribo"},
    {"TRINIDAD AND TOBAGO","Port of Spain"},
    {"URUGUAY","Montevideo"}, {"VENEZUELA","Caracas"},
  };
} ///:~
```

Esto es simplemente un array bidimensional de **cadena de caracteres**[5]. He aquí una simple prueba que utiliza los métodos **rellenar( )** y generadores:

```
//: c09:PruebaRellenar.java
import com.bruceeckel.util.*;
import java.util.*;

public class PruebaRellenar {
  static Generator sg =
    new Arrays2.GeneradorStringAleatorio(7);
  public static void main(String[] args) {
    List lista = new ArrayList();
    Colecciones2.rellenar(lista, sg, 25);
    System.out.println(lista + "\n");
    List lista2 = new ArrayLista();
    Colecciones2.rellenar(list2,
      Colecciones2.capitales, 25);
    System.out.println(lista2 + "\n");
    Set conjunto = new HashSet();
    Colecciones2.rellenar(conjunto, sg, 25);
    System.out.println(conjunto + "\n");
    Map m = new HashMap();
    Colecciones2.rellenar(m, Colecciones2.rsp, 25);
    System.out.println(m + "\n");
    Map m2 = new HashMap();
    Colecciones2.rellenar(m2,
      Colecciones2.geografia, 25);
    System.out.println(m2);
  }
} ///:~
```

Con estas herramientas se pueden probar de forma sencilla los diversos contenedores, rellenándolos con datos que interesen.

---

[5] Estos datos se encontraron en Internet, y después se procesaron mediante un programa en Python (véase *http://www.Python.org*).

Desventaja de los contenedores: tipo desconocido

El "inconveniente" de usar los contenedores de Java es que se pierde información de tipos cuando se introduce un objeto en un contenedor. Esto ocurre porque el programador de la clase contenedora no sabía qué tipo específico se iba a guardar en el contenedor, y construirlo de forma que sólo almacene un tipo concreto haría que éste dejase se ser una herramienta de propósito general. Así, el contenedor simplemente almacena referencias a **Object**, que es la raíz de todas las clases, y así se puede guardar cualquier tipo. (Por supuesto, esto no incluye los tipos primitivos, puesto que éstos no se heredan de nada.) Esto es una solución grandiosa excepto por:

1. Dado que se deja de lado la información de tipos al introducir un objeto en el contenedor, no hay restricción relativa al tipo de objetos que se pueden introducir en un contenedor, incluso si se desea que almacene exclusivamente, por ejemplo, gatos. Alguien podría colocar un perro sin ningún tipo de problema en ese contenedor.

2. Dado que se pierde la información de tipos, lo único que sabe el contenedor es que guarda referencias a objetos. Es necesario hacer una conversión al tipo correcto antes de usar esas referencias.

En el lado positivo, puede decirse que Java no permitirá un *uso erróneo* de los objetos que se introduzcan en un contenedor. Si se introduce un perro en un contenedor de gatos y después se intenta manipular el contenido como si de un gato se tratara, se obtendrá una excepción de tiempo de ejecución en el momento de extraer la referencia al perro del contenedor de gatos e intentar hacerle una conversión a gato.

He aquí un ejemplo utilizando el contenedor básico **ArrayList**. Los principiantes pueden pensar que **ArrayList** es "un array que se expande a sí mismo automáticamente". Usar un **ArrayList** es muy directo: se crea, se introducen objetos usando **add( )** y posteriormente se extraen con **get( )** haciendo uso del índice —exactamente igual que se haría con un array pero sin los corchetes[6].

ArrayList también tiene un método **size( )** que permite saber cuántos elementos se han añadido de forma que uno no se pasará de largo sin querer, generando una excepción.

En primer lugar, se crean las clases **Gato** y **Perro**:

```
//: c09:Gato.java
public class Gato {
  private int numGato;
  Gato(int i) { numGato = i; }
  void escribir() {
    System.out.println("Gato #" + numGato);
  }
```

[6] Éste es un punto en el que sería muy indicada la sobrecarga de operadores.

```
} ///:~


//: c09:Perro.java
public class Perro {
  private int numPerro;
  Perro(int i) { numPerro = i; }
  void escribir() {
    System.out.println("Perro #" + numPerro);
  }
} ///:~
```

Se introducen en el contenedor **Gatos** y **Perros**, y después se extraen:

```
//: c09:GatosYPerros.java
// Ejemplo simple de contenedores.
import java.util.*;

public class GatosYPerros {
  public static void main(String[] args) {
    ArrayList Gatos = new ArrayList();
    for(int i = 0; i < 7; i++)
      Gatos.add(new Gato(i));
    // Añadir perros o gatos no es ningún problema:
    Gatos.add(new Perro(7));
    for(int i = 0; i < Gatos.size(); i++)
      ((Gato)Gatos.get(i)).escribir();
    // El perro sólo se detecta en tiempo de ejecución
  }
} ///:~
```

Las clases **Gato** y **Perro** son distintas —no tienen nada en común, excepto que ambas son **Objeto**s. (Si no se dice explícitamente de qué clase se está heredando, se considera que se está haciendo directamente de **Object**.) Dado que **ArrayList** guarda **Objeto**s, no sólo se pueden introducir objetos **Gato** mediante el método **add( )** de **ArrayList**, sino que también es posible añadir objetos **Perro** sin que se dé ninguna queja ni en tiempo de compilación ni en tiempo de ejecución. Cuando se desea recuperar eso que se piensa que son objetos **Gato** utilizando el método **get( )** de **ArrayList**, se obtiene una referencia a un objeto que debe convertirse previamente a **Gato**. Por tanto, es necesario envolver toda la expresión con paréntesis para forzar la evaluación de la conversión antes de invocar al método **escribir( )** de **Gato**, pues si no se obtendrá un error sintáctico. Posteriormente, en tiempo de ejecución, cuando se intente convertir el objeto **Perro** en **Gato** se obtendrá una excepción.

Esto es más que sorprendente. Es algo que puede ser causa de errores muy difíciles de encontrar. Si se tiene una parte (o varias) de un programa insertando objetos en un contenedor, y se descubre mediante una expresión en un único fragmento del programa que se ha ubicado algún objeto de tipo erróneo en un contenedor, es necesario averiguar posteriormente dónde se dio la inserción errónea.

Lo positivo del asunto, es que es posible y conveniente empezar a programar con clases contenedoras estandarizadas, en vez de buscar la especificidad y complejidad del código.

En ocasiones funciona de cualquier modo

Resulta que en algunos casos parece que todo funciona correctamente sin tener que hacer una conversión al tipo original. Hay un caso bastante especial: la clase **String** tiene ayuda extra del compilador para hacer que funcione correctamente. En cualquier ocasión que el compilador espere un objeto **String** y no obtenga uno, invocará automáticamente al método **toString( )** definido en **Object** y que puede ser superpuesto por cualquier clase Java. Este método produce el objeto **String** deseado, y que es posteriormente utilizado allí donde se necesitaba un **String**.

Por tanto, todo lo que se necesita es construir objetos que superpongan el método **toString( )**, como se ve en el ejemplo siguiente:

```
//: c09:Raton.java
// Superponiendo toString().
public class Raton {
  private int numRaton;
  Raton(int i) { numRaton = i; }
  // Superponer Object.toString():
  public String toString() {
    return "Este es el Raton #" + numRaton;
  }
  public int obtenerNumero() {
    return numRaton;
  }
} ///:~


//: c09:TrabajarCualquierModo.java
// En determinados casos, las cosas simplemente
// parecen funcionar correctamente.
import java.util.*;

class TrampaRaton {
  static void capturar(Object m) {
    Raton raton = (Raton)m; // Conversión desde Object
    System.out.println("Raton: " +
      raton.obtenerNumero());
  }
}

public class TrabajarCualquierModo {
  public static void main(String[] args) {
```

```
    ArrayList ratones = new ArrayList();
    for(int i = 0; i < 3; i++)
      ratones.add(new Raton(i));
    for(int i = 0; i < ratones.size(); i++) {
      // No es necesaria conversión, se invoca automáticamente:
      // a Object.toString()
      System.out.println(
        "Raton libre: " + ratones.get(i));
      TrampaRaton.Capturar(ratones.get(i));
    }
  }
} ///:~
```

Podemos ver que en **Ratón** se ha superpuesto **toString( )**. En el segundo bucle **for** del método **main( )** se encuentra la sentencia:

```
System.out.println("Raton libre: " + ratones.get(i));
```

Después del signo "+" el compilador espera un objeto **de tipo String**. El método **get( )** produce un **Object**, por lo que para lograr la cadena de caracteres deseada, el compilador llama implícitamente a **toString( )**. Desgraciadamente, esta especie de magia sólo funciona con cadenas de caracteres: no está disponible para ningún otro tipo.

El segundo enfoque para ocultar la conversión se encuentra dentro de **TrampaRaton**. El método **capturar( )** no acepta un **Ratón** sino un **Objeto**, que después se convierte en **Ratón**. Esto es bastante presuntuoso, por supuesto, pues al aceptar un **Objeto**, se podría pasar al método cualquier cosa. Sin embargo, si la conversión es incorrecta —se pasa un tipo erróneo —se genera una excepción en tiempo de ejecución. Esto no es tan bueno como una comprobación en tiempo de compilación, pero sigue siendo robusto. Fíjese que en el uso de este método:

```
TrampaRaton.Capturar(ratones.get(i));
```

no es necesaria ninguna conversión.

Hacer un **ArrayList** consciente de los tipos

No debería abandonar aún este asunto. Una solución a toda prueba pasa por crear una nueva clase haciendo uso de **ArrayList**, que sólo acepte un determinado tipo, y produzca sólo ese determinado tipo:

```
//: c09:listaRaton.java
// Un ArrayList consciente de los tipos.
import java.util.*;

public class ListaRaton {
```

```
  private ArrayList lista = new ArrayList();
  public void aniadir(Mouse m) {
    lista.add(m);
  }
  public Raton obtener(int indice) {
    return (Raton)lista.get(indice);
  }
  public int tamanio() { return lista.size(); }
} ///:~
```

He aquí una prueba del nuevo contenedor:

```
//: c09:PruebaListaRaton.java
public class PruebaListaRaton {
  public static void main(String[] args) {
    ListaRaton mice = new ListaRaton();
    for(int i = 0; i < 3; i++)
      ratones.add(new Raton(i));
    for(int i = 0; i < ratones.size(); i++)
      TrampaRaton.capturar(ratones.get(i));
  }
} ///:~
```

Esto es similar al ejemplo anterior, excepto en que la nueva clase **ListaRaton** tiene un miembro **privado** de tipo **ArrayList**, y métodos iguales a los de **ArrayList**. Sin embargo, no acepta y produce **Objetos** genéricos, sino sólo objetos **Ratón**.

Nótese que si por el contrario se hubiera heredado **ListaRaton** de **ArrayList**, el método **aniadir(Raton)** simplemente habría sobrecargado el **add(Object)** existente, y seguiría sin haber restricción alguna en el tipo de objetos que se podrían añadir. Por consiguiente, el **ListaRaton** se convierte en un *sustituto de* **ArrayList**, que lleva a cabo algunas actividades antes de pasar la responsabilidad (véase *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com*).

Dado que un objeto de tipo **ListaRaton** únicamente aceptará un **Ratón**, si se dice:

```
ratones.aniadir(new Paloma());
```

se obtendrá un mensaje de error *en tiempo de compilación*. Este enfoque, aunque es más tedioso desde el punto de vista del código, indicará inmediatamente si se está usando un tipo de manera inadecuada.

Nótese que no es necesaria ninguna conversión al usar **get( )** —siempre es un **Ratón**.

Tipos parametrizados

Este tipo de problema no está aislado —hay numerosos casos en los que es necesario crear nuevos tipos basados en otros tipos, y en los que es útil tener información de tipo específica en tiempo de compilación. Éste es el concepto de *tipo parametrizado*. En C++, esto se soporta directamente por el

lenguaje gracias a las *plantillas*. Es probable que una versión futura de Java soporte alguna variación de los tipos parametrizados; actualmente simplemente se crean clases similares a **ListaRaton**.

Iteradores

En cualquier clase contenedora, hay que tener una forma de introducir y extraer elementos. Después de todo, éste es el primer deber de un contenedor —almacenar elementos. En el **ArrayList**, **add( )** es la forma de insertar objetos, y **get( )** es *una* de las formas de extraer objetos. **ArrayList** es bastante flexible —se puede seleccionar cualquier cosa en cualquier momento, y seleccionar múltiples elementos a la vez, utilizando índices diferentes.

Si se desea empezar a pensar en un nivel superior, hay un inconveniente: hay que conocer el tipo exacto de contenedor para poder usarlo. Esto podría no parecer malo a primera vista, pero ¿qué ocurre si se empieza a usar **ArrayList**, y más adelante en el programa se descubre que debido al uso que se le está dando al contenedor sería más eficiente usar un **LinkedList** en su lugar? O suponga que se desea escribir un fragmento de código genérico independiente del tipo de contenedor con el que trabaje, ¿cómo podría hacerse de forma que éste pudiera usarse en distintos tipos de contenedores sin tener que reescribir ese código?

El concepto de *iterador* puede usarse para lograr esta abstracción. Un iterador es un objeto cuyo trabajo es moverse a lo largo de una secuencia de objetos y seleccionar cada objeto de esa secuencia sin que el programador cliente tenga que saber u ocuparse de la estructura subyacente de esa secuencia. Además, un iterador es lo que generalmente se llama un objeto "ligero": un objeto fácil de crear. Por esa razón, a menudo uno encontrará restricciones extrañas para los iteradores; por ejemplo, algunos de ellos sólo pueden moverse en una dirección.

El **Iterator** de Java es un ejemplo de un iterador con este tipo de limitaciones. No hay mucho que se pueda hacer con él salvo:

1. Pedir a un contendor que porporcione un **iterador** utilizando un método denominado **iterador ( )**. Este **Iterador** estará listo para devolver el primer elemento de la secuencia en la primera llamada a su método **next( )**.
2. Conseguir el siguiente objeto de la secuencia con **next( )**.
3. Ver si *hay* más objetos en la secuencia con **hasNext( )**.
4. Eliminar el último elemento devuelto por el iterador con **remove( )**.

Esto es todo. Es una implementación simple de un iterador, pero aún con ello es potente (y hay un **ListIterator** más sofisticado para las **Listas**). Para ver cómo funciona, podemos volver a echar un vistazo al programa **PerrosYGatos.java** visto antes en este capítulo. En la versión original, se usaba el método **get( )** para seleccionar cada elemento, pero en la versión modificada siguiente se usa un iterador:

```
//: c09:PerrosYGatos2.java
// Contenedor simple con Iterador.
import java.util.*;
```

```
public class GatosYPerros2 {
  public static void main(String[] args) {
    ArrayList gatos = new ArrayList();
    for(int i = 0; i < 7; i++)
      gatos.add(new Gato(i));
    Iterator e = gatos.iterator();
    while(e.hasNext())
      ((Gato)e.next()).escribir();
  }
} ///:~
```

Se puede ver que las últimas líneas usan ahora un **Iterador** para recorrer la secuencia, en vez de un bucle **for**. Con el **Iterador**, no hay que preocuparse por el número de elementos del contenedor. Son los métodos **hasNext( )** y **next( )** los que se encargan de esto por nosotros.

Como otro ejemplo, considérese la creación de un método de impresión de propósito general:

```
//: c09:LaberintoHamster.java
// Usando un Iterador.
import java.util.*;

class Hamster {
  private int numHamster;
  Hamster(int i) { numHamster = i; }
  public String toString() {
    return "Este el el Hamster #" + numHamster;
  }
}

class Escritura {
  static void escribirTodo(Iterador e) {
    while(e.hasNext())
      System.out.println(e.next());
  }
}

public class LaberintoHamster {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 3; i++)
      v.add(new Hamster(i));
    escribir.escribirTodo(v.iterador());
  }
} ///:~
```

Echemos un vistazo a **escribirTodo( )**. Nótese que no hay información relativa al tipo de secuencia. Todo lo que se tiene es un **Iterador**, y esto es todo lo que hay que saber de la secuencia: que se puede conseguir el siguiente objeto, y que se puede saber cuándo se llega al final. Esta idea de tomar un contenedor de objetos y recorrerlo para llevar a cabo una operación sobre cada uno es potente, y se verá con detenimiento a lo largo de este libro.

El ejemplo es incluso más genérico, pues implícitamente usa el método **Object.toString( )**. El método **println( )** está sobrecargado para todos los tipos primitivos además de **Object**; en cada caso se produce automáticamente un **cadena de caracteres** llamando al método **toString( )** apropiado.

Aunque no es necesario, se puede ser más explícito usando una conversión, que tiene el efecto de llamar a **toString( )**:

```
System.out.println((String)e.next());
```

En general, sin embargo, se deseará hacer algo más que llamar a métodos de **Object**, por lo que habrá que enfrentarse de nuevo al problema de la conversión de tipos. Hay que asumir que se tiene un **Iterador** para una secuencia de un tipo particular en el que se está interesado, y que hay que convertir los objetos resultantes a ese tipo (consiguiendo una excepción en tiempo de ejecución si se hace mal).

Recursividad involuntaria

Dado que los contenedores estándar de Java son heredados (como con cualquier otra clase) de **Object**, contienen un método **toString( )**. Éste ha sido superpuesto de forma que pueden producir una representación **String** de sí mismos, incluyendo los objetos que guardan. Dentro de **ArrayList**, por ejemplo, el método **toString( )** recorre los elementos del **ArrayList** y llama a **toString( )** para cada uno de ellos. Supóngase que se desea imprimir la dirección de la clase. Parece que tiene sentido hacer simplemente referencia a **this** (son los programadores de C++, en particular, los que más tienden a esto).

```
//: c09:RecursividadInfinita.java
// Recursividad accidental.
import java.util.*;

public class RecursividadInfinita {
  public String toString() {
    return " Recursividad infinita direccion: "
      + this + "\n";
  }
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++)
      v.add(new RecursividadInfinita());
    System.out.println(v);
  }
} ///:~
```

Si simplemente se crea un objeto **RecursividadInfinita** y luego se imprime, se consigue una secuencia interminable de excepciones. Esto también es cierto si se ubican los objetos **RecursividadInfinita** en un **ArrayList** y se imprime ese **ArrayList** como se ha mostrado. Lo que está ocurriendo es una conversión de tipos automática a cadenas de caracteres. Al decir:

```
" Recursividadinfinita direccion: " + this
```

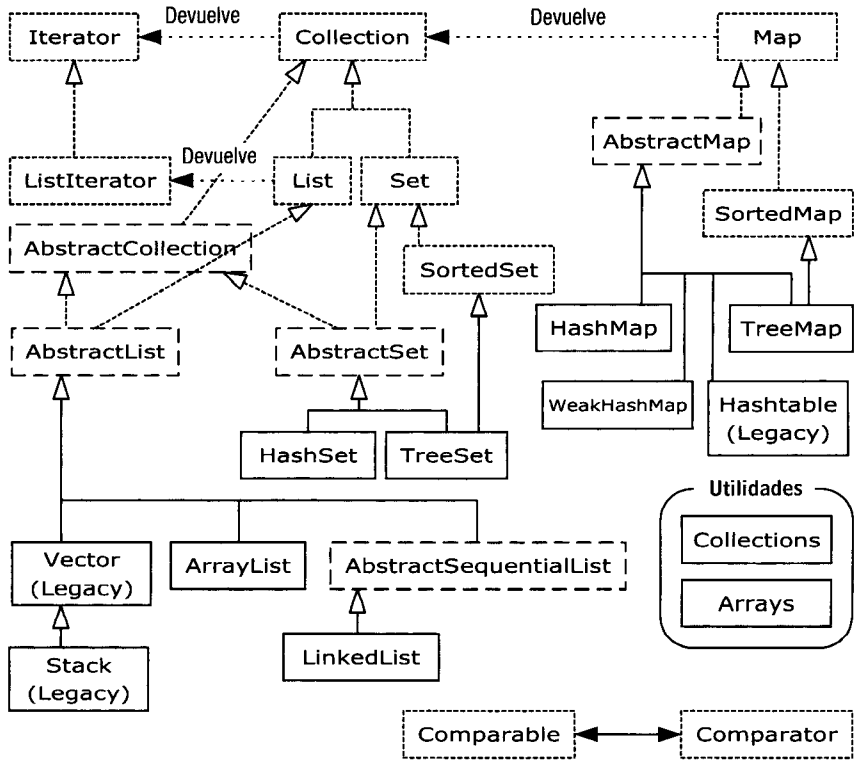
el compilador ve **cadena de carecteres** seguido de un "+" y algo que no es un **cadena de caracteres**, por lo que intenta convertir **this** a **cadena de caracteres**. Hace esta conversión llamando al método **toString( )**, lo cual produce una llamada recursiva.

Si verdaderamente se desea imprimir en este caso la dirección del objeto, la solución es llamar al método **toString( )**,de object que hace justamente eso.

Por tanto, en vez de decir **this**, se debería decir **super.toString( )**. (Esto sólo funciona si se ha heredado directamente de **Object**, o si ninguna de las clases padre ha superpuesto el método **toString( )**.)

Taxonomía de contenedores

Las **colecciones** y **mapas** pueden implementarse de distintas formas, en función de las necesidades de programación. Nos será de ayuda echar un vistazo al diagrama de los contenedores de Java 2:



Este diagrama puede ser un poco cargante al principio, pero se verá que realmente sólo hay tres componentes contenedores: **Map**, **List** y **Set**, y sólo hay dos o tres implementaciones de cada uno (habiendo una versión preferida, generalmente). Cuando vemos esto, los contenedores no son tan intimidadores.

Las cajas punteadas representan **interfaces**, las cajas a trazos representan clases **abstractas**, y las cajas continuas son clases normales (concretas). Las flechas de líneas de puntos indican que una clase particular implementa una **interfaz** (o en el caso de una clase **abstracta**, que se implementa parcialmente una **interfaz**). Las flechas continuas muestran que una clase puede producir objetos de la clase a la que apunta la flecha. Por ejemplo, cualquier **Collection** puede producir un **Iterator**, mientras que una **List**, puede producir un **ListIterator** (además de un **Iterator** normal, dado que **List** se hereda de **Collection**).

Las interfaces relacionadas con el almacenamiento de objetos son **Collection**, **List**, **Set** y **Map**. Idealmente, se escribirá la mayor parte del código para comunicarse con estas interfaces, y el único lugar en el que se especifica el tipo concreto que se está usando es en el momento de su creación. Por tanto, se puede crear un objeto **List** como éste:

```
List x = new LinkedList();
```

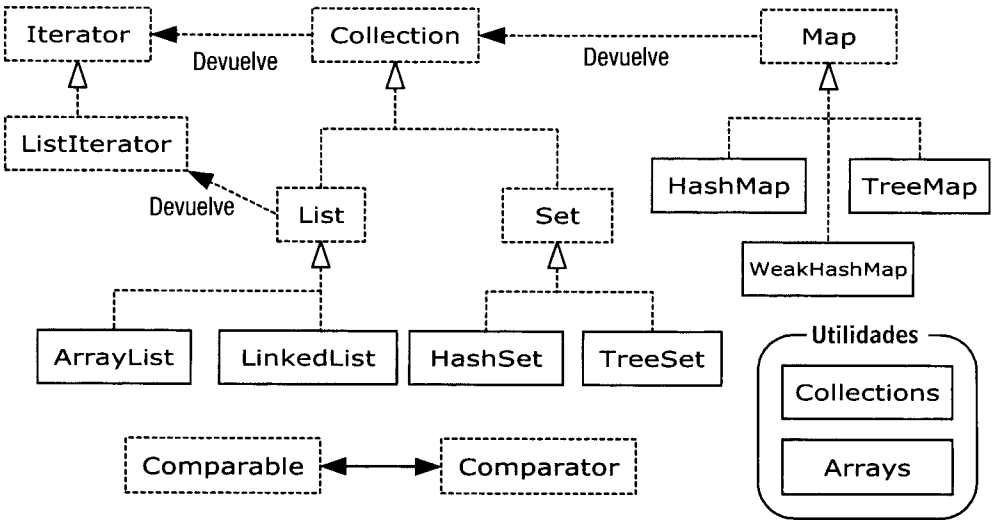
Por supuesto, se puede decidir que **x** sea una **LinkedList** (en vez de un objeto **List** genérico) y acarrear la información de tipos junto con **x**. La belleza (y la intención) de utilizar la **interfaz** es que si se desea cambiar la implementación, todo lo que hay que hacer es cambiarla en el instante de su creación, así:

```
List x = new ArrayList();
```

El resto del código puede mantenerse intacto (parte de esta generalidad puede lograrse con iteradores).

En la jerarquía de clases, se pueden ver varias clases cuyos nombres empiezan por "**Abstract**", y esto podría parecer un poco confuso al principio. Son simplemente herramientas que implementan parcialmente una interfaz particular. Si uno estuviera construyendo, por ejemplo, su propio **Set**, no empezaría con la interfaz **Set** para luego implementar todos los métodos, sino que se heredaría de **AbstractSet** haciendo el mínimo trabajo necesario para construir la nueva clase. Sin embargo, la biblioteca de contenedores contiene funcionalidad necesaria para satisfacer prácticamente todas las necesidades en todo momento. Por tanto, para nuestros propósitos, se puede ignorar cualquier clase que comience con "**Abstract**".

Por consiguiente, cuando se mire al diagrama, sólo hay que fijarse en las **interfaces** de la parte superior del diagrama y las clases concretas (las cajas de trazo continuo). Generalmente se harán objetos de clases concretas, se hará conversión hacia arriba a la **interfaz** correspondiente, y después se usará esa **interfaz** durante todo el resto del código. Además, no es necesario considerar los elementos antiguos al escribir código nuevo. Por consiguiente, el diagrama puede simplificarse enormemente a:



Ahora sólo incluye las interfaces y clases que se encontrarán normalmente, además de los elementos en los que se centrará el presente capítulo.

He aquí un ejemplo que rellena un objeto **Collection** (representado aquí con un **ArrayList**) con objetos **String**, y después imprime cada elemento del objeto **Collection**:

```
//: c09:ColeccionSencilla.java
// Un ejemplo sencillo usando las Colecciones de Java 2.
import java.util.*;

public class ColeccionSencilla {
  public static void main(String[] args) {
    // Conversión hacia arriba porque queremos
    // trabajar sólo con aspectos de Colección
    Collection c = new ArrayList();

    for(int i = 0; i < 10; i++)
      c.add(Integer.toString(i));
    Iterator it = c.iterator();
    while(it.hasNext())
      System.out.println(it.next());
  }
} ///:~
```

La primera línea del método **main( )** crea un objeto **ArrayList** y después hace una conversión hacia arriba a **Collection**. Dado que este ejemplo sólo usa los métodos **Collection**, funcionaría con cualquier objeto de una clase heredada de **Collection**, pero **ArrayList** es el objeto de tipo **Collection** con el que se suele trabajar.

El método **add( )**, como sugiere su nombre, pone un nuevo elemento en el objeto **Collection**. Sin embargo, la documentación establece claramente que **add( )** "asegura que este contenedor contiene el elemento especificado". Esto es para que sea compatible con el significado de **Set**, que añade el elemento sólo si no está ya ahí. Con un **ArrayList**, o cualquier tipo de **List**, **add( )** siempre significa "introducirlo", porque a las **Listas** no les importa la existencia de duplicados.

Todas las **Colecciones** pueden producir un **Iterador** mediante su método **iterator( )**. Aquí se crea un **Iterador** y luego se usa para recorrer la **Colección**, imprimiendo cada elemento.

Funcionalidad de la **Collection**

La siguiente tabla muestra todo lo que se puede hacer con una **Collection** (sin incluir los métodos que vienen automáticamente con **Object**), y por consiguiente, todo lo que se puede hacer con un **Set** o un **List**. (**List** también tiene funcionalidad adicional.) Los objetos **Map** no se heredan de **Collection**, y se tratarán de forma separada.

boolean add(Object)	Asegura que el contenedor tiene el parámetro. Devuelve falso si no añade el parámetro. (Éste es un método "opcional", descrito más adelante en este capítulo.)
boolean addAll(Collection)	Añade todos los elementos en el parámetro. Devuelve **verdadero** si se añadió alguno de los elementos. ("Opcional.")
void clear()	Elimina todos los elementos del contenedor. ("Opcional.")
boolean contains(Object)	**verdadero** si el contenedor almacena el parámetro.
boolean containsAll(Collection)	**verdadero** si el contenedor guarda todos los elementos del parámetro.
boolean isEmpty()	**verdadero** si el contenedor no tiene elementos.
Iterator iterator()	Devuelve un **Iterador** que se puede usar para recorrer los elementos del contenedor.
boolean remove(Object)	Si el parámetro está en el contenedor, se elimina una instancia de ese elemento. Devuelve **verdadero** si se produce alguna eliminación. ("Opcional.")
boolean removeAll(Collection)	Elimina todos los elementos contenidos en el parámetro. Devuelve **verdadero** si se da alguna eliminación. ("Opcional.")
boolean retainAll(Collection)	Mantiene sólo los elementos contenidos en el parámetro (una "intersección" en teoría de conjuntos). Devuelve **verdadero** si se dio algún cambio. ("Opcional.")

int size()	Devuelve el número de elementos del contenedor.
Object[] toArray()	Devuelve un array que contenga todos los elementos del contenedor.
Object[] toArray(Object[] a)	Devuelve un array que contiene todos los elementos del contenedor, cuyo tipo es el del array **a** y no un simple **Object** (hay que convertir el array al tipo correcto).

Nótese que no hay función **get( )** para selección de elementos por acceso al azar. Eso es porque **Collection** también incluye **Set**, que mantiene su propia ordenación interna (y por consiguiente convierte en carente de sentido el acceso aleatorio). Por consiguiente, si se desean examinar todos los elementos de una **Collection** hay que usar un iterador; es la única manera de recuperar las cosas.

El ejemplo siguiente demuestra todos estos métodos. De nuevo, éstos trabajan con cualquier objeto heredado de **Collection**, pero se usa un **ArrayList** como "mínimo común denominador":

```
//: c09:Coleccion1.java
// Cosas que se pueden hacer con todas las Colecciones.
import java.util.*;
import com.bruceeckel.util.*;

public class Coleccion1 {
  public static void main(String[] args) {
    Collection c = new ArrayList();
    Colecciones2.rellenar(c,
      Colecciones2.Paises, 10);
    c.add("diez");
    c.add("once");
    System.out.println(c);
    // Hacer un array a partir de Lista:
    Object[] array = c.toArray();
    // Hacer un String a partir de una Lista:
    String[] str =
      (String[])c.toArray(new String[1]);
    // Encontrar los elementos max y min; esto
    // conlleva distintas cosas en función de cómo
    // se implemente la interfaz Comparable:
    System.out.println("Collections.max(c) = " +
      Collections.max(c));
    System.out.println("Collections.min(c) = " +
      Collections.min(c));
    // Añadir una Colección a otra Colección
    Collection c2 = new ArrayList();
    Colecciones2.rellenar(c2,
```

```
        Colecciones2.Paises, 10);
    c.addAll(c2);
    System.out.println(c);
    c.remove(CapitalesPaises.pares[0][0]);
    System.out.println(c);
    c.remove(CapitalesPaises.pares[1][0]);
    System.out.println(c);
    // Quitar todos los elementos de la colección
    // pasada como parámetro:
    c.removeAll(c2);
    System.out.println(c);
    c.addAll(c2);
    System.out.println(c);
    // ¿Es un elemento de esta colección?
    String val = CapitalesPaises.pares[3][0];
    System.out.println(
      "c.contains(" + val  + ") = "
      + c.contains(val));
    // ¿Es una Colección de esta Colección?
    System.out.println(
      "c.containsAll(c2) = "+ c.containsAll(c2));
    Collection c3 = ((List)c).subList(3, 5);
    // Mantener todos los elementos que están tanto en
    // c2 como en c3 (intersección de conjuntos):
    c2.retainAll(c3);
    System.out.println(c);
    // Quitar todos los elementos
    // de c2 que también aparecen en c3:
    c2.removeAll(c3);
    System.out.println("c.isEmpty() = " +
      c.isEmpty());
    c = new ArrayList();
    Colecciones2.rellenar(c,
      Colecciones2.Paises, 10);
    System.out.println(c);
    c.clear(); // Eliminar todos los elementos
    System.out.println("despues c.clear():");
    System.out.println(c);
  }
} ///:~
```

Los objetos de tipo **ArrayList** se crean conteniendo distintos conjuntos de datos y hacen conversión hacia arriba a objetos **Collection**, por lo que está claro que no se está usando nada más que la interfaz **Collection**. El método **main( )** usa ejercicios simples para mostrar todos los métodos de **Collection**.

La sección siguiente describe las diversas implementaciones de **List**, **Set** y **Map** e indica en cada caso (con un asterisco) cuál debería ser la selección por defecto. El lector se dará cuenta de que *no* se han incluido las clases antiguas **Vector**, **Stack** y **Hashtable** porque en todos los casos son preferibles las clases Contenedoras de Java 2.

Funcionalidad de la interfaz **List**

La clase **List** básica es bastante fácil de usar, como ya se ha visto con **ArrayList**. Aunque la mayoría de veces simplemente se usará **add( )** para insertar objetos, **get( )** para sacarlos todos a la vez el **iterator( )** para lograr un **Iterador** para la secuencia, también hay un conjunto de otros métodos que podrían ser útiles.

Además, hay, de hecho, dos tipos de objetos **List**: el **ArrayList** básico que destaca entre los elementos de acceso aleatorio, y la mucho más potente **LinkedList** (que no fue diseñada para un acceso aleatorio rápido, pero que tiene un conjunto de métodos mucho más generales).

int size( )	Devuelve el número de elementos del contenedor.
List (interfaz)	El orden es la secuencia más importante de una **List**; promete mantener los elementos en una secuencia determinada. **List** añade varios métodos a **Collection** que permiten la inserción y eliminación de elementos en el medio de un objeto **List**. (Esto sólo está recomendado en el caso de **LinkedList**.) Un objeto **List** producirá un **ListIterator**, gracias al cual se puede recorrer la lista en ambas direcciones, además de insertar y eliminar elementos en medio de un objeto **List**.
ArrayList*	Un objeto **List** implementado con un array. Permite acceso aleatorio rápido a los elementos, pero es lento si se desea insertar o eliminar elementos del medio de una lista. Debería usarse **ListIterator** sólo para recorridos hacia adelante y hacia atrás de un **ArrayList**, pero no para insertar y eliminar elementos, lo que es caro si se compara con **LinkedList**.
LinkedList	Proporciona acceso secuencial óptimo, con inserciones y borrados en la parte central de la **List**. Es relativamente lenta para acceso al azar. (En este caso hay que usar **ArrayList**.) También tiene **addFirst( )**, **addLast( )**, **getFirst( )**, **getLast( )**, **removeFirst( )**, y **removeLast( )** (que no están definidos en ninguna de las interfaces o clases base) para permitir su uso como si se tratara de una pila, una cola o una bicola.

Los métodos del ejemplo siguiente cubren cada uno un grupo de actividades: las cosas que puede hacer toda lista (**basicTest( )**), recorrido con un **Iterador** (**iterMotion( )**) frente a cambiar cosas con un **Iterador** (**iterManipulation( )**), la observación de los efectos de manipulación de objetos **List** (**testVisual( )**), y operaciones disponibles únicamente para objetos **LinkedLists**.

```
//: c09:Lista1.java
// Cosas que se pueden hacer con Listas.
import java.util.*;
import com.bruceeckel.util.*;

public class Lista1 {
  public static List rellenar(List a) {
    Colecciones2.Paises.reset();
    Colecciones2.fill(a,
      Colecciones2.Paises, 10);
    return a;
  }
  static boolean b;
  static Object o;
  static int i;
  static Iterator it;
  static ListIterator lit;
  public static void PruebaBasica(List a) {
    a.add(1, "x"); // Añadir en la posición 1
    a.add("x"); // Añadir al final
    // Añadir una colección:
    a.addAll(rellenar(new ArrayList()));
    // Añadir una colección empezando en la posición 3:
    a.addAll(3, rellenar(new ArrayList()));
    b = a.contains("1"); // ¿Está ahí?
    // ¿Está la colección entera ahí?
    b = a.containsAll(rellenar(new ArrayList()));
    // Las listas permiten acceso al azar, lo que es barato
    // para ArrayList, y caro para LinkedList:
    o = a.get(1); // Recuperar el objeto de la posición 1
    i = a.indexOf("1"); // Devolver el índice de un objeto
    b = a.isEmpty(); // ¿Hay algún elemento dentro?
    it = a.iterator(); // Iterator ordinario
    lit = a.listIterator(); // ListIterator
    lit = a.listIterator(3); // Empezar en la posición 3
    i = a.lastIndexOf("1"); // Última coincidencia
    a.remove(1); // Eliminar la posición 1
    a.remove("3"); // Eliminar este objeto
    a.set(1, "y"); // Poner la posición 1 a "y"
    // Mantener todo lo que esté en los parámetros
    // (la intersección de los dos conjuntos):
    a.retainAll(rellenar(new ArrayList()));
    // Quitar todo lo que esté en los parámetros:
    a.removeAll(rellenar(new ArrayList()));
    i = a.size(); // ¿Cuán grande es?
```

```
    a.clear(); // Quitar todos los elementos
  }
  public static void movimientoIterador(List a) {
    ListIterator it = a.listIterator();
    b = it.hasNext();
    b = it.hasPrevious();
    o = it.next();
    i = it.nextIndex();
    o = it.previous();
    i = it.previousIndex();
  }
  public static void manejoIterador(List a) {
    ListIterator it = a.listIterator();
    it.add("47");
    // Moverse a un elemento después de add():
    it.next();
    // Quitar el elemento recién creado:
    it.remove();
    // Moverse a un elemento después de remove():
    it.next();
    // Cambiar el elemento recién creado:
    it.set("47");
  }
  public static void pruebaVisual(List a) {
    System.out.println(a);
    List b = new ArrayList();
    rellenar(b);
    System.out.print("b = ");
    System.out.println(b);
    a.addAll(b);
    a.addAll(rellenar(new ArrayList()));
    System.out.println(a);
    // Insertar, eliminar y reemplazar elementos
    // usando un ListIterator:
    ListIterator x = a.listIterator(a.size()/2);
    x.add("one");
    System.out.println(a);
    System.out.println(x.next());
    x.remove();
    System.out.println(x.next());
    x.set("47");
    System.out.println(a);
    // Recorrer la lista hacia atrás:
    x = a.listIterator(a.size());
    while(x.hasPrevious())
```

```
        System.out.print(x.previous() + " ");
      System.out.println();
      System.out.println("Fin de prueba Visual");
    }
    // Hay cosas que sólo pueden hacer los objetos
    // LinkedLists:
    public static void pruebaLinkedList() {
      LinkedList ll = new LinkedList();
      rellenar(ll);
      System.out.println(ll);
      // Tratarlo como una pila, apilar:
      ll.addFirst("uno");
      ll.addFirst("dos");
      System.out.println(ll);
      // Cómo "mirar a hurtadillas" a la cima de la pila:
      System.out.println(ll.getFirst());
      // Cómo desapilar de la pila:
      System.out.println(ll.removeFirst());
      System.out.println(ll.removeFirst());
      // Tratarlo como una cola sacando elementos
      // por el final:
      System.out.println(ll.removeLast());
      // ¡Con las operaciones de arriba es una bicola!
      System.out.println(ll);
    }
    public static void main(String[] args) {
      // Hacer y rellenar una nueva lista cada vez:
      pruebaBasica(rellenar(new LinkedList()));
      pruebaBasica(rellenar(new ArrayList()));
      movimientoIterador(rellenar(new LinkedList()));
      manejoIterador(rellenar(new ArrayList()));
      manejoIterador(rellenar(new LinkedList()));
      PruebaVisual(rellenar(new ArrayList()));
      PruebaVisual(rellenar(new LinkedList()));
      pruebaLinkedList();
    }
} ///:~
```

En **pruebaBasica( )** y **movimientoIterador( )** las llamadas se hacen simplemente para mostrar la sintaxis correcta, capturando, pero no usando, el valor de retorno. En algunos casos, no se captura el valor de retorno, puesto que no suele usarse. Sería conveniente echar un vistazo a la documentación relativa a la utilización completa de cada uno de estos métodos, en la documentación *en línea* de *java.sun.com*.

Construir una pila a partir de un objeto **LinkedList**

A una pila se le suele denominar contenedor "*last-in, first-out*" o LIFO (el último en entrar es el primero en salir). Es decir, lo que se meta ("apilar") lo último en la pila, será lo primero que se puede extraer ("desapilar"). Como ocurre con el resto de contenedores de Java, lo que se mete y extrae son **Objetos**, por lo que se debe convertir lo que se extrae, a menos que se esté haciendo uso del comportamiento de **Object**.

El **LinkedList** tiene métodos que implementan directamente la funcionalidad de una pila, por lo que también se puede usar una **LinkedList** en vez de hacer una clase pila. Sin embargo, puede ser que una clase pila se comporte mejor:

```
//: c09:PilaL.java
// Haciendo una pila a partir de una LinkedList.
import java.util.*;
import com.bruceeckel.util.*;

public class PilaL {
  private LinkedList lista = new LinkedList();
  public void apilar(Object v) {
    lista.addFirst(v);
  }
  public Object cima() { return list.getFirst(); }
  public Object desapilar() {
    return lista.removeFirst();
  }
  public static void main(String[] args) {
    PilaL Pila = new PilaL();
    for(int i = 0; i < 10; i++)
      Pila.apilar(Colecciones2.Paises.next());
    System.out.println(Pila.cima());
    System.out.println(Pila.cima());
    System.out.println(Pila.desapilar());
    System.out.println(Pila.desapilar());
    System.out.println(Pila.desapilar());
  }
} ///:~
```

Si sólo se quiere el comportamiento de la pila, es inapropiado hacer uso aquí de la herencia, pues produciría una clase con todo el resto de métodos de **LinkedList** (se verá más tarde que los diseñadores de la biblioteca de Java 1.0 cometieron este error con **Stack**).

Construir una cola a partir de un objeto **LinkedList**

Una *cola* es un contenedor *"first-in, first-out"* FIFO (el primero en entrar es el primero en salir). Es decir, se introducen los elementos por un extremo y se sacan por el otro. Por tanto, los elementos se extraerán en el mismo orden en que fueron introducidos. **LinkedList** tiene métodos para soportar el comportamiento de una cola, que pueden usarse en una clase **Cola**:

```
//: c09:Cola.java
// Haciendo una cola a partir de un objeto LinkedList.
import java.util.*;

public class Cola {
  private LinkedList lista = new LinkedList();
  public void poner(Object v) { lista.addFirst(v); }
  public Object quitar() {
    return lista.removeLast();
  }
  public boolean estaVacia() {
    return lista.isEmpty();
  }
  public static void main(String[] args) {
    Cola cola = new Cola();
    for(int i = 0; i < 10; i++)
      cola.poner(Integer.toString(i));
    while(!cola.estaVacia())
      System.out.println(cola.quitar());
  }
} ///:~
```

También se puede crear fácilmente una *bicola* (cola de dos extremos) a partir de un objeto **LinkedList**. Esta es como una cola, pero se puede tanto insertar como eliminar elementos por ambos extremos de la misma.

Funcionalidad de la interfaz **Set**

Set tiene exactamente la misma interfaz que **Collection**, por lo que no hay ninguna funcionalidad como la que hay con los dos objetos **List**. Sin embargo, **Set** es exactamente igual a **Collection**; simplemente tiene un comportamiento distinto. (Éste es el uso ideal de la herencia y del polimorfismo: expresar comportamiento distinto.) Un **Set** es un objeto **Collection** que rehúsa guardar más de una instancia de cada valor de objeto (lo que constituye el "valor" del objeto es más complejo, como veremos).

Set (interfaz)	Cada elemento que se añada al objeto **Set** debe ser único; si no es así, **Set** no añade el elemento duplicado. Los **Objects** añadidos a un **Set** deben implementar **equals( )** para establecer la unicidad de los objetos. **Set** tiene exactamente el mismo interfaz que **Collection**. La interfaz **Set** no garantiza que mantenga sus elementos en ningún orden particular.
HashSet*	En los objetos **Set** en los que el tiempo de búsqueda sea importante, los **objectos** deben definir también un **HashCode( )**.
TreeSet	Un **Set** ordenado respaldado por un árbol. De esta forma, se puede extraer una secuencia ordenada de objeto **Set**.

El ejemplo siguiente *no* muestra todo lo que se puede hacer con un objeto Set, dado que la interfaz es el misma que la de **Collection**, y así se vio en el ejemplo anterior. Lo que sí hace es demostrar el comportamiento que convierte a un objeto **Set** en único:

```
//: c09:Conjunto1.java
// Cosas que se pueden hacer con conjuntos.
import java.util.*;
import com.bruceeckel.util.*;

public class Conjunto1 {
  static Colecciones2.GeneradorString gen =
    Colecciones2.Paises;
  public static void pruebaVisual(Set a) {
    Colecciones2.rellenar(a, gen.inicializar(), 10);
    Colecciones2.rellenar(a, gen.inicializar(), 10);
    Colecciones2.rellenar(a, gen.inicializar(), 10);
    System.out.println(a); // ¡Sin repeticiones!
    // Añadir otro conjunto a éste:
    a.addAll(a);
    a.add("uno");
    a.add("uno");
    a.add("uno");
    System.out.println(a);
    // Buscar algo:
    System.out.println("a.contains(\"uno\"): " +
      a.contains("uno"));
  }
  public static void main(String[] args) {
    System.out.println("HashSet");
    pruebaVisual(new HashSet());
    System.out.println("TreeSet");
    pruebaVisual(new TreeSet());
  }
```

```
} ///:~
```

Al objeto **Set** se le añaden valores duplicados, pero al imprimirlo se verá que el objeto **Set** solamente ha aceptado una instancia de cada uno de los valores.

Al ejecutar este programa veremos que el orden mantenido por el **HashSet** es distinto del de **TreeSet**, dado que cada uno tiene una manera de almacenar los elementos a fin de localizarlos después. (**TreeSet** los mantiene ordenados, mientras que **HashSet** usa una función de *conversión de clave*, diseñada específicamente para lograr búsquedas rápidas.) Cuando uno crea sus propios tipos, debe ser consciente de que un objeto **Set** necesita una manera de mantener un orden de almacenamiento, lo que significa que hay que implementar la interfaz **Comparable**, y definir el método **compareTo( )**. He aquí un ejemplo:

```
//: c09:Conjunto2.java
// Metiendo un tipo propio en un Conjunto.
import java.util.*;

class MiTipo implements Comparable {
  private int i;
  public MiTipo(int n) { i = n; }
  public boolean equals(Object o) {
    return
      (o instanceof MiTipo)
      && (i == ((MiTipo)o).i);
  }
  public int hashCode() { return i; }
  public String toString() { return i + " "; }
  public int compareTo(Object o) {
    int i2 = ((MiTipo)o).i;
    return (i2 < i ? -1 : (i2 == i ? 0 : 1));
  }
}

public class Conjunto2 {
  public static Set rellenar(Set a, int tamanio) {
    for(int i = 0; i < tamanio; i++)
      a.add(new MiTipo(i));
    return a;
  }
  public static void prueba(Set a) {
    rellenar(a, 10);
    rellenar(a, 10); // Intentar introducir duplicados
    rellenar(a, 10);
    a.addAll(rellenar(new TreeSet(), 10));
    System.out.println(a);
  }
```

```
  public static void main(String[] args) {
    prueba(new HashSet());
    prueba(new TreeSet());
  }
} ///:~
```

La forma de las definiciones de **equals( )** y **HashCode( )** se describirá más adelante en este mismo capítulo. Hay que definir un **equals( )** en ambos casos, pero el **HashCode( )** es absolutamente necesario sólo si se ubicara la clase en un **Hash.Set** (lo que es probable, pues ésta suele ser la primera opción a la hora de implementar un **Set**). Sin embargo, como estilo de programación debería superponerse **HashCode( )** siempre al superponer **equals( )**. Analizaremos el proceso detalladamente más adelante en este capítulo.

En el método **compareTo( )**, fíjese que *no* se usó la forma "simple y obvia" **return i-i2**. Aunque éste es un error de programación común, sólo funcionaría correctamente si **i** y **i2** fueran **enteros** "sin signo" (si Java *tuviera* una palabra clave "unsigned", pero no la tiene). No funciona para el **entero** con signo de Java, ya que no es lo suficientemente grande como para representar la diferencia de dos **enteros** con signo. Si **i** es un entero grande positivo y **j** es un entero grande negativo, **i-j** causará un desbordamiento y devolverá un error negativo, lo cual no funcionará.

Conjunto ordenado (**SortedSet**)

Si se tiene un objeto **SortedSet** (del que sólo está diponible el **TreeSet**), se garantiza que los elementos se ordenarán permitiendo proporcionar funcionalidad adicional con estos métodos de la interfaz **SortedSet**:

Comparator comparator( ): Devuelve un objeto **Comparator** utilizado para este objeto **Set**, o **null** en el caso de ordenamiento natural.

Object first( ): Devuelve el elemento menor.

Object last( ): Devuelve el elemento mayor.

SortedSet subSet(desdeElemento, hastaElemento): Proporciona una vista de este objeto **Set** desde el elemento de **desdeElemento**, incluido, hasta **hastaElemento**, excluido.

SortedSet headSet(hastaElemento): Devuelve una vista de este objeto **Set** con los elementos menores a **hastaElemento**.

SortedSet tailSet(desdeElemento): Devuelve una vista de este objeto **Set** con elementos mayores o iguales a **desdeElemento**.

Funcionalidad Map

Un objeto **ArrayList** permite hacer una selección a partir de una secuencia de objetos usando un número, por lo que en cierta forma asigna números a los objetos. Pero, ¿qué ocurre si se desea se-

leccionar un objeto de una secuencia siguiendo algún otro criterio? Una pila es un ejemplo de esto: su criterio de selección es "el último elemento insertado en la pila". Un giro contudente de esta idea de "seleccionar a partir de una secuencia" se le denomina un *mapa*, un diccionario o un *array asociativo*. Conceptualmente, parece un objeto **ArrayList**, pero en vez de buscar los objetos usando un número, se buscan utilizando ¡*otro objeto*! Éste suele ser un proceso clave en un programa.

Este concepto se presenta en Java como la interfaz **Map**. El método **put(Object clave, Object valor)** añade un valor (el elemento deseado), y le asocia una clave (el elemento con el que buscar). **get(Object clave)** devuelve el valor a partir de la clave correspondiente. También se puede probar un **Map** para ver si contiene una clave o valor con **containsKey( )** y **containsValue( )**.

La biblioteca estándar de Java tiene dos tipos diferentes de **objetos Map**: **HashMap** y **TreeMap**. Ambos tienen la misma interfaz (dado que ambos implementan **Map**), pero difieren claramente en la eficiencia. Si se observa lo que hace un **get( )**, parecerá bastante lento hacerlo buscando a través de la clave (por ejemplo) de un **ArrayList**. Es aquí donde un **HashMap** acelera considerablemente las cosas. En vez de hacer una búsqueda lenta de la clave, usa un valor especial denominado *código de tipo hash*. Ésta es una manera de tomar cierta información del objeto en cuestión y convertirlo en un **entero** "relativamente único" para ese objeto. Todos los objetos de Java pueden producir un código de tipo hash, y **HashCode( )** es un método de la clase raíz **Object**. Un **HashMap** toma un **hashCode( )** del objeto y lo utiliza para localizar rápidamente la clave. Esto redunda en una mejora dramática de rendimiento[7].

Map	Mantiene asociaciones clave-valor (pares), de forma que se puede buscar un valor usando una clave.
HashMap*	La implementación basada en una tabla de tipo hash. (Utilizar esto en vez de **Hashtable**.) Proporciona rendimiento constante en el tiempo para insertar y localizar pares. El rendimiento se puede ajustar mediante constructores que permiten especificar la *capacidad* y el *factor de carga* de la tabla de tipo hash.
TreeMap	Implementación basada en un árbol. Cuando se vean las claves de los pares, se ordenarán (en función de **Comparable** o **Comparator**, que se discutirán más tarde). La clave de un **TreeMap** es que se logran resultados en orden. **TreeMap** es el único objeto **Map** con un método **subMap( )**, que permite devolver un fragmento de árbol.

En ocasiones también es necesario conocer los detalles de cómo funciona la conversión hash, por lo que le echaremos un vistazo un poco después.

El ejemplo siguiente usa el método **Colecciones2.rellenar( )** y los conjuntos de datos de prueba definidos previamente:

[7] Si estas mejoras de velocidad siguen sin ajustarse a las necesidades de rendimiento pretendidas, se puede acelerar aún más la búsqueda en la tabla escribiendo un **Mapa** personalizado, y adaptándolo a tipos particulares que eviten retrasos debidos a conversiones hacia y desde **Object**. Para llegar a niveles de rendimiento aún mejores, los entusiastas de la velocidad pueden usar el *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition*, de Donald Knuth, para reemplazar las listas de *cubos* de recorrido por arrays que presenten dos beneficios adicionales: pueden optimizarse para características de almacenamiento de disco y pueden ahorrar la mayoría del tiempo de creación y recolección de basura de registros individuales.

```
//: c09:Mapa1.java
// Cosas que se pueden hacer con Mapas.
import java.util.*;
import com.bruceeckel.util.*;

public class Mapa1 {
  static Colecciones2.GeneradorParString geo =
    Colecciones2.geografia;
  static Colecciones2.GeneradorParStringAleatorio
    rsp = Colecciones2.rsp;
  // Produciendo un conjunto de claves:
  public static void escribirClaves(Map m) {
    System.out.print("Tamaño = " + m.size() +", ");
    System.out.print("Claves: ");
    System.out.println(m.keySet());
  }
  // Produciendo una Colección de valores:
  public static void escribirValores(Map m) {
    System.out.print("Valores: ");
    System.out.println(m.values());
  }
  public static void prueba(Map m) {
    Colecciones2.rellenar(m, geo, 25);
    // Mapa tiene comportamiento 'de conjunto' para las claves:
    Colecciones2.rellenar(m, geo.reset(), 25);
    escribirClaves(m);
    escribirValores(m);
    System.out.println(m);
    String clave = CapitalesPaises.pares[4][0];
    String valor = CapitalesPaises.pares[4][1];
    System.out.println("m.containsKey(\"" + clave +
      "\"): " + m.containsKey(clave));
    System.out.println("m.get(\"" + clave + "\"): "
      + m.get(clave));
    System.out.println("m.containsValue(\""
      + valor + "\"): " +
      m.containsValor(valor));
    Map m2 = new TreeMap();
    Coleciones2.rellenar(m2, rsp, 25);
    m.putAll(m2);
    escribirClaves(m);
    Clave = m.keySet().iterator().next().toString();
    System.out.println("Primera clave del mapa: "+clave);
    m.remove(Clave);
    escribirClaves(m);
```

```
    m.clear();
    System.out.println("m.isEmpty(): "
      + m.isEmpty());
    Colecciones2.rellenar(m, geo.inicializar(), 25);
    // Las operaciones en el Conjunto cambian el Mapa:
    m.keySet().removeAll(m.keySet());
    System.out.println("m.isEmpty(): "
      + m.isEmpty());
  }
  public static void main(String[] args) {
    System.out.println("Prueba HashMap");
    prueba(new HashMap());
    System.out.println("Prueba TreeMap");
    prueba(new TreeMap());
  }
} ///:~
```

Los métodos **escribirClaves( )** y **escribirValores( )** no son simples utilidades, sino que también demuestran cómo producir vistas **Collection** de un **Mapa**. El método **keySet( )** devuelve un **Conjunto** respaldado por las claves de un **Mapa**. A **values( )** se le da un tratamiento similar, al producir un objeto **Colección** que contiene todos los valores del **Mapa**. (Nótese que las claves deben ser únicas, aunque los valores pueden contener duplicados.) Dado que estas **Colecciones** están respaldadas por el **Mapa**, cualquier cambio en una **Colección** se reflejará en el **Mapa** asociado.

El resto del programa proporciona ejemplos sencillos de cada operación de **Mapa**, y prueba cada tipo de **Mapa**.

Como ejemplo de uso de un objeto **HashMap**, considérese un programa que compruebe cómo funciona el método de Java **Math.random( )**. De manera ideal, produciría una distribución perfecta de números al azar, pero para probarlo es necesario generar un conjunto de números al azar y contar cuántos caen en cada subrango. Para esto es perfecto un objeto **HashMap**, puesto que asocia objetos con objetos (en este caso, el objeto valor contiene el número producido por **Math.random( )** junto con la cantidad de veces que aparece ese número):

```
//: c09:Estadisticos.java
// Demostración sencilla de HashMap.
import java.util.*;

class Contador {
  int i = 1;
  public String toString() {
    return Integer.toString(i);
  }
}

class Estadisticos {
```

```
  public static void main(String[] args) {
    HashMap hm = new HashMap();
    for(int i = 0; i < 10000; i++) {
      // Producir un número entre 0 y 20:
      Integer r =
        new Integer((int)(Math.random() * 20));
      if(hm.containsKey(r))
        ((Counter)hm.get(r)).i++;
      else
        hm.put(r, new Contador());
    }
    System.out.println(hm);
  }
} ///:~
```

En el método **main( )**, cada vez que se genera un número aleatorio, se envuelve en un objeto **Integer** de forma que pueda usarse la referencia con el objeto **HashMap**. (No se puede usar una primitiva con un contenedor, sólo una referencia a objeto.) El método **containsKey( )** comprueba si la clave ya está en el contenedor. (Es decir, ¿se ha encontrado ya el número?) En caso afirmativo, el método **get( )** produce el valor asociado para la clave, que en este caso es un objeto **Contador**. El valor **i** del contenedor se incrementa para indicar que se ha encontrado una ocurrencia más de este número al azar en particular.

Si no se ha encontrado aún la clave, el método **put( )** ubicará un nuevo par clave —valor en el objeto **HashMap**. Desde que el **Contador** inicializa automáticamente su variable **i** a uno cuando se crea, indica la primera ocurrencia de este número al azar en concreto.

Para mostrar el **HashMap**, simplemente se imprime. El método **toString( )** de **HashMap** se mueve por todos los pares clave-valor y llama a **toString( )** para cada uno. El **Integer.toString( )** está predefinido, y se puede ver el **toString( )** para **Contador**. La salida de una ejecución (a la que se han añadido saltos de línea) es:

```
{ 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=480,   4=489,   3-509,   2=503,   1= 475,
  0=505}
```

Uno se podría plantear la necesidad de la clase **Contador**, que parece que ni siquiera tuviera la funcionalidad de la clase envoltorio **Integer**. ¿Por qué no usar **int** o **Integer**? Bien, no se puede usar un tipo primitivo **entero** porque ninguno de los contenedores puede guardar nada que no sean referencias a **Objetos**. Después de ver los contenedores, puede que las clases envoltorio comiencen a tomar un mayor significado, puesto que no se puede introducir ningún tipo primitivo en los contenedores. Sin embargo, lo único que se *puede* hacer con los envoltorios de Java es inicializarlos a un valor particular y leer ese valor. Es decir, no hay forma de cambiar un valor una vez que se ha creado un objeto envoltorio. Esto hace que el envoltorio **Integer** sea totalmente inútil para solucionar nuestro problema, por lo que nos vemos forzados a crear una nueva clase para satisfacer esta necesidad.

Mapa ordenado (Sorted Map)

Si se tiene un objeto **SortedMap** (de la que **TreeMap** es lo único disponible) se garantiza que las claves se ordenarán de manera que con los siguientes métodos de la interfaz **SortedMap** se proporcione la siguiente funcionalidad:

Comparator comparator( ): Devuelve el comparador usado para este **Mapa**, o **null** para ordenación natural.

Object firstKey( ): Devuelve la clave más baja.

Object lastKey( ): Devuelve la clave más alta.

SortedMap subMap(desdeClave, hastaClave): Produce una vista de este **Mapa** con las claves que van desde **desdeClave** incluida, hasta **hastaClave** excluida.

SortedMap headMap(hastaClave): Produce una vista de este **Mapa** con las claves menores de **hastaClave**.

SortedMap tailMap(desdeClave): Produce una vista de este **Mapa** con las claves mayores o iguales que **desdeClave**.

Hashing y códigos de hash

En el ejemplo anterior, se usó una clase de biblioteca estándar (**Integer**) como clave del objeto **HashMap**. Funcionaba bien como clave, porque tenía todo lo que necesitaba. Pero se puede caer en una trampa común con objetos **HashMap** al crear clases propias para usar como claves. Por ejemplo, considérese un sistema de predicción del tiempo que une objetos **Meteorólogo** con objetos **Predicción**. Parece bastante directo—se crean dos clases, y se usa **Meteorólogo** como clave y **Predicción** como valor:

```
//: c09:DetectorPrimavera.java
// Parece creible, pero no funciona.
import java.util.*;

class Meteorologo {
  int numeroMet;
  Meteorologo(int n) { numeroMet = n; }
}

class Prediccion {
  boolean oscurecer = Math.random() > 0.5;
  public String toString() {
    if(oscurecer)
      return "¡Seis semanas mas de invierno!";
    else
```

```
        return "primavera temprana!";
    }
}

public class DetectorPrimavera {
    public static void main(String[] args) {
        HashMap hm = new HashMap();
        for(int i = 0; i < 10; i++)
            hm.put(new Meteorologo(i), new Prediccion());
        System.out.println("hm = " + hm + "\n");
        System.out.println(
            "Buscando prediccion para Meteorologo #3:");
        Meteorologo gh = new Meteorologo(3);
        if(hm.containsKey(gh))
            System.out.println((Prediccion)hm.get(gh));
        else
            System.out.println("Clave no encontrada: " + gh);
    }
} ///:~
```

A cada **Meteorólogo** se le da un número de identidad, de forma que se puede buscar una **Predicción** en el objeto **HashMap** diciendo: "Dame la **Predicción** asociada al **Meteorólogo** número 3." La clase **Predicción** contiene un **valor lógico** que se inicializa utilizando **Math.random()**, y un **toString( )** que interpreta el resultado. En el método **main( )**, se rellena un objeto **HashMap** con objetos de tipo **Meteorólogo** y sus **Predicciones** asociadas. Se imprime el **HashMap** de forma que se puede ver que ha sido rellenada. Después, se usa un **Meteorólogo** con el número de identidad 3 para buscar la predicción de **Meterólogo** número 3 (que como puede verse debe estar en el **Mapa**).

Parece lo suficientemente simple, pero no funciona. El problema es que **Meteorólogo** se hereda de la clase raíz común **Object** (que es lo que ocurre si no se especifica una clase base, pues todas las clases se heredan en última instancia de **Object**). Es el método **hashCode( )** de **Object** el que se usa para generar el código hash de cada objeto, y por defecto, usa únicamente la dirección del objeto. Por tanto, la primera instancia de **Meteorólogo(3)** *no* produce un código de hash igual al código de hash de la segunda instancia de **Meteorólogo(3)** que se intentó usar en la búsqueda.

Se podría pensar que todo lo que se necesita hacer es escribir una superposición adecuada de **hashCode( )**. Pero seguirá sin funcionar hasta que se haya hecho otra cosa: superponer el método **equals( )** que también es parte de **Object**. Este método se usa en **HashMap** al intentar determinar si una clave es igual a alguna de las claves de la tabla. De nuevo, el **Object.equals( )** por defecto simplemente compara direcciones de objetos, por lo que un **Meteorólogo(3)** no es igual a otro **Meteorólogo(3)**.

Por tanto, para utilizar nuestras clases como claves en un objeto **HashMap**, es necesario superponer tanto **hashCode( )** como **equals( )**, como se muestra en la siguiente solución al problema descrito:

```
//: c09:DetectorPrimavera2.java
// Una clase usada como clave de un HashMap
```

```
// debe superponer hashCode() y equals().
import java.util.*;

class Meteorologo2 {
  int numeroMet;
  Meteorologo2(int n) { numeroMet = n; }
  public int hashCode() { return numeroMet; }
  public boolean equals(Object o) {
    return (o instanceof Meteorologo2)
      && (numeroMet == ((Meteorologo2)o).numeroMet);
  }
}

public class DetectorPrimavera2 {
  public static void main(String[] args) {
    HashMap hm = new HashMap();
    for(int i = 0; i < 10; i++)
      hm.put(new Meteorologo2(i),new Prediccion());
    System.out.println("hm = " + hm + "\n");
    System.out.println(
      "Buscando una prediccion para el Meteorologo #3:");
    Meteorologo2 gh = new Meteorologo2(3);
    if(hm.containsKey(gh))
      System.out.println((Prediccion)hm.get(gh));
  }
} ///:~
```

Fíjese que este ejemplo usa la clase **Predicción** del ejemplo anterior, de forma que debe compilarse primero **DetectorPrimavera.java** o se obtendrá un error de tiempo de compilación al intentar compilar **DetectorPrimavera2.java**.

Meteorologo2.hashCode( ) devuelve el número de meteorólogo como identificador. En este ejemplo, el programador es responsable de asegurar que no existirá el mismo número de ID en dos meteorólogos. No se requiere que **hashCode( )** devuelva un identificador único (algo que se entenderá mejor algo más adelante en este capítulo), pero el método **equals( )** debe ser capaz de determinar estrictamente si dos objetos son o no equivalentes.

Incluso aunque parece que el método **equals( )** sólo hace comprobaciones para ver si el parámetro es una instancia de **Meteorólogo2** (utilizando la palabra clave **instanceof**, que se explica completamente en el Capítulo 12), **instanceof** hace, de hecho, silenciosamente una segunda comprobación, para ver si el objeto es **null**, dado que **instanceof** devuelve **falso** si el parámetro de la izquierda es **null**. Asumiendo que sea del tipo correcto y no **null**, la comparación se basa en el **numeroMet** actual. Esta vez, cuando se ejecute el programa se verá que produce la salida correcta.

Al crear tu propia clase para usar en **HashSet**, hay que prestar atención a los mismos aspectos que cuando se usa como clave en un **HashMap**.

Comprendiendo **hashCode()**

El ejemplo de arriba es sólo un inicio de cara a solucionar el problema correctamente. Muestra que si no se superponen **hashCode( )** y **equals( )** para la clave, la estructura de datos de hash (**HashSet** o **HashMap**) no podrán manejar la clave adecuadamente. Sin embargo, para lograr una solución correcta al problema hay que entender lo que está ocurriendo dentro de la estructura de datos.

En primer lugar, considérese la motivación que hay tras el proceso hash se desea buscar un objeto utilizando otro objeto. Pero se puede lograr esto con un objeto **TreeSet** o un objeto **TreeMap**. También es posible implementar tu propio **Mapa**. Para hacerlo, se suministrará el método **Map.entrySet( )** para producir un conjunto de objetos **Map.Entry**. **MPar** también se definirá como el nuevo tipo de **Map.Entry**. Para poder ubicarlo en un objeto **TreeSet**, debe implementar **equals( )** y ser **Comparable**:

```
//: c09:MPar.java
// Un Map implementado con ArrayLists.
import java.util.*;

public class MPar
implements Map.Entry, Comparable {
  Object clave, valor;
  MPar(Object k, Object v) {
    clave = k;
    valor = v;
  }
  public Object obtenerClave() { return clave; }
  public Object obtenerValor() { return valor; }
  public Object ponerValor(Object v){
    Object resultado = valor;
    valor = v;
    return resultado;
  }
  public boolean equals(Object o) {
    return clave.equals(((MPar)o).clave);
  }
  public int compareTo(Object rv) {
    return ((Comparable)clave).compareTo(
      ((MPar)rv).clave);
  }
} ///:~
```

Fíjese que a las comparaciones sólo les interesan las claves, por lo que se aceptan perfectamente valores duplicados.

El ejemplo siguiente usa un **Mapa** utilizando un par de objetos **ArrayList**:

```
//: c09:MapaLento.java
// Un Mapa implementado con ArrayList.
```

```
import java.util.*;
import com.bruceeckel.util.*;

public class MapaLento extends AbstractMap {
  private ArrayList
    claves = new ArrayList(),
    valores = new ArrayList();
  public Object put(Object clave, Object valor) {
    Object resultado = get(clave);
    if(!claves.contains(clave)) {
      claves.add(clave);
      valores.add(valor);
    } else
      valores.set(claves.indexOf(clave), valor);
    return result;
  }
  public Object get(Object clave) {
    if(!claves.contains(clave))
      return null;
    return valores.get(claves.indexOf(clave));
  }
  public Set entrySet() {
    Set entradas = new HashSet();
    Iterator
      ki = claves.iterator(),
      vi = valores.iterator();
    while(ki.hasNext())
      entradas.add(new MPar(ki.next(), vi.next()));
    return entradas;
  }
  public static void main(String[] args) {
    MapaLento m = new MapaLento();
    Colecciones2.rellenar(m,
      Colecciones2.geografia, 25);
    System.out.println(m);
  }
} ///:~
```

El método **put( )** simplemente ubica las claves y valores en los objetos de tipo **ArrayList** correspondientes. En el método **main( )** se carga un **MapaLento** y después se imprime para que se vea cómo funciona.

Esto muestra que no es complicado producir un nuevo tipo de **Mapa**. Pero como sugiere el nombre, un **MapaLento** no es muy rápido, por lo que probablemente no se usará si se dispone de alguna alternativa. El problema se da en la búsqueda de la clave: no hay orden, por lo que se usa una simple búsqueda lineal, que es la forma más lenta de buscar algo.

El único motivo de utilizar hash es la velocidad: el uso de hash permite que la búsqueda se realice rápidamente. Dado que el cuello de botella se encuentra en la velocidad de búsqueda de la clave, una de las soluciones al problema podría ser mantener las claves ordenadas y después realizar la búsqueda usando **Collections.binarySearch( )** (un ejercicio planteado al final de este capítulo consiste en esto).

El uso de hash va aún más lejos diciendo que todo lo que se desea hacer es almacenar la clave *en algún sitio,* de forma que pueda encontrarse rápidamente. Como se ha visto en este capítulo, la estructura más rápida en la que almacenar un grupo de elementos es un array, por lo que se usará éste para representar la información de claves (fíjese en el detalle de que dijimos "información de claves" y no las propias claves). También se ha visto en este capítulo que un array, una vez asignado, no se puede redimensionar, por lo que se tiene un problema: se desea poder almacenar cualquier número de valores en el **Mapa**, pero si el número de claves viene fijado por el tamaño del array ¿qué se puede hacer?

La respuesta es que el array no almacenará las claves. Desde la clave del objeto, se derivará un número que se indexará al array. Este número es el código de *hash*, producido por el método **HashCode( )** (en informática, a esta función se le denominaría *función de hash*) definido en **Object**, y presumiblemente superpuesto en la propia clase. Para solucionar el problema del array de tamaño fijo, se permite que más de una clave produzca el mismo índice. Es decir, puede haber *colisiones*. Debido a esto, no importa lo grande que sea el array, puesto que cada objeto caerá en algún lugar del mismo.

Por tanto el proceso de buscar un valor comienza computando el código de hash y usándolo como índice del array. Si se pudiera garantizar que no se dieran colisiones (lo que podría ser posible si se tuviera un número fijo de valores) entonces se tendría una *función de hash perfecta*, pero éste es un caso especial. En el resto de ocasiones, las colisiones se manejan mediante *encadenado externo*: el array no apunta directamente a un valor, sino a una lista de valores. Estos valores se buscan de manera lineal utilizando el método **equals( )**. Por supuesto, este aspecto de la búsqueda es mucho más lento, pero si la función de hash fuera buena sólo habría unos pocos valores (como máximo) en cada posición. Por tanto, en vez de buscar por toda la lista, se salta directamente a una *ranura* en la que simplemente hay que comprobar unas pocas entradas para encontrar el valor. Esto es mucho más rápido, siendo la razón la rapidez de **HashMap**.

Conociendo lo básico del uso de claves hash, es posible implementar un **Mapa** simple con técnicas de hash:

```
//: c09:HashMapSencillo.java
// Una demostración del Mapa que utiliza hash.
import java.util.*;
import com.bruceeckel.util.*;

public class HashMapSencillo extends AbstractMap {
  // Elegir un número primo para el tamaño de la tabla de
  // hash, para lograr una distribución uniforme:
  private final static int SZ = 997;
  private LinkedList[] cubo= new LinkedList[SZ];
```

```
public Object put(Object clave, Object valor) {
  Object resultado = null;
  int indice = clave.hashCode() % SZ;
  if(indice < 0) indice = -indice;
  if(cubo[indice] == null)
    cubo[indice] = new LinkedList();
  LinkedList pares = cubo[index];
  MPar par = new MPar(clave, valor);
  ListIterator it = pares.listIterator();
  boolean encontrado = false;
  while(it.hasNext()) {
    Object iPar = it.next();
    if(iPar.equals(par)) {
      resultado = ((MPar)iPar).obtenerValor();
      it.set(par); // Reemplazar viejo con nuevo
      encontrado = true;
      break;
    }
  }
  if(!encontrado)
    cubo[indice].add(par);
  return result;
}
public Object get(Object clave) {
  int indice = clave.hashCode() % SZ;
  if(indice < 0) indice = -indice;
  if(cubo[indice] == null) return null;
  LinkedList pares = cubo[indice];
  MPar parear = new MPar(clave, null);
  ListIterator it = pares.listIterator();
  while(it.hasNext()) {
    Object iPar = it.next();
    if(iPar.equals(parear))
      return ((MPar)iPar).obtenerValor();
  }
  return null;
}
public Set entrySet() {
  Set entradas = new HashSet();
  for(int i = 0; i < cubo.length; i++) {
    if(cubo[i] == null) continue;
    Iterator it = cubo[i].iterator();
    while(it.hasNext())
      entradas.add(it.next());
  }
```

```
    return entradas;
  }
  public static void main(String[] args) {
    SimpleHashMap m = new SimpleHashMap();
    Colecciones2.rellenar(m,
      Colecciones2.geografia, 25);
    System.out.println(m);
  }
} ///:~
```

Dado que a las "posiciones" de una tabla de hash se les suele llamar cubos *(buckets)*, se llama **cubo** al array que representa esta tabla. Para promocionar la distribución uniforme, el número de cubos suele ser un número primo. Fíjese que es un array de **LinkedList**, que automáticamente soporta colisiones —cada nuevo *elemento* simplemente se añade al final de la lista.

El valor de retorno de **put( )** es **null** o, si la clave ya estaba en la lista, el valor viejo asociado a esa clave. El valor de retorno es **resultado**, que se inicializa a **null**, pero si se descubre una clave en la lista, se asigna **resultado** a esa clave.

Tanto para **put( )** como **get( )**, lo primero que ocurre es que se invoca a **hashCode( )** para lograr la clave, y se fuerza a que el resultado sea positivo. Después, se hace que encaje en el array **cubo** utilizando el operador módulo y el tamaño del array. Si esa posición vale **null**, quiere decir que no hay elementos a los que el hash condujo a esa posición, por lo que se crea una nueva **LinkedList** para guardar los objetos. Sin embargo, el proceso normal es mirar en la lista para ver si hay duplicados, y si los hay, se pone el valor viejo en **resultado** y el nuevo valor reemplaza al viejo. El indicador **encontrado** mantiene un seguimiento de si se ha encontrado un par clave-valor antiguo, y si no, se añade el nuevo par al final de la lista.

En **get( )**, se verá código muy similar al contenido en **put( )**, pero más simple. Se calcula el índice en el array **cubo**, y si existe una **LinkedList** en ella, se busca la coincidencia en la misma.

El método **entrySet( )** debe encontrar y recorrer todas las listas, añadiéndolas al **Set**. Una vez que se ha creado este método, se prueba el **Map** rellenándolo de valores para después imprimirlos.

Factores de rendimiento de **HashMap**

Para entender los aspectos es necesaria cierta terminología:

Capacidad: El número de posiciones de la tabla.

Capacidad inicial: Número de posiciones en la creación de la tabla.

HashMap y **HashSet**: Tienen constructores que permiten especificar su capacidad inicial.

Tamaño: Número de elementos actualmente en la tabla.

Factor de carga: tamaño/capacidad. Un factor de carga 0 es una tabla vacía; 0,5 es una tabla medio llena, etc. Una tabla con una densidad de carga alta tendrá pocas colisiones, por lo que es óptima para búsquedas e inserciones (pero ralentizará el proceso de recorrido con

un iterador). **HashMap** y **HashSet** tienen constructores que permiten especificar el factor de carga, lo que significa que cuando se llega a este factor de carga el contenedor incrementará automáticamente la capacidad (el número de posiciones) doblándola, y redistribuirá los objetos existentes en el nuevo conjunto de posiciones (a esta operación se la llama *redistribución de las claves hash*).

El factor de carga por defecto utilizado por **HashMap** es 0,75 (no hace una redistribución de claves hash hasta que 3/4 de la tabla están llenos). Esto parece ofrecer un buen equilibrio entre costo en espacio y costo en tiempo. Un factor de carga mayor disminuye el espacio que la tabla necesita, pero incrementa el coste de búsqueda, que es importante pues lo que más se hará son búsquedas (incluyendo **get( )** y **put( )**.

Si se sabe que se almacenarán muchas entradas en un **HashMap**, crearlo con una capacidad inicial apropiadamente grande, evitará la sobrecarga debida a la redistribución automática.

Superponer el método **hashCode( )**

Ahora que se entiende lo que está involucrado en la función del objeto **HashMap**, los aspectos involucrados en la escritura de método **hashCode( )** tendrán más sentido.

En primer lugar, no se tiene control de la creación del valor actual utilizado para indexar el array de elementos. Este valor depende de la capacidad del objeto **HashMap** particular, y esa capacidad cambia dependiendo de lo lleno que esté el contenedor y de cuál sea el factor de carga. El valor producido por el método **hashCode( )** se procesará después para crear el índice de los elementos (en **SimpleHashMap** el cálculo es un módulo por el tamaño del array de elementos).

El factor más importante de cara a la creación de un método **hashCode( )** es que, independientemente de cuándo se llame a **hashCode( )**, produzca el mismo valor para cada objeto cada vez que se le llame. Si se tuviera un objeto que produce un valor **hashCode( )** cuando se invoca a **put( )** para introducirlo en un **HashMap**, y otro durante un **get( )**, no se podrían retirar los objetos. Por tanto, si el **hashCode( )** depende de datos mutables del objeto, el usuario debe ser consciente de que cambiar los datos producirá una clave diferente generando un **hashCode( )** distinto.

Además, probablemente *no* se desee generar un **hashCode( )** que se base en una única información de un objeto —en particular, el valor de **this** genera un **hashCode( )** malo, pues no se puede generar una nueva clave idéntica a la usada al hacer **put( )** con el par clave-valor original. Éste era el problema de **DetectorPrimavera.java**, dado que la implementación por defecto de **hashCode( )** *sí* que utiliza la dirección del objeto. Por tanto, se deseará utilizar información del objeto que lo identifique de manera significativa.

En la clase **String**, puede verse un ejemplo de esto. Los objetos **String** tienen la característica especial de que si un programa tiene varios objetos **String** con secuencias de caracteres idénticas, todos esos objetos **String** hacen referencia a la misma memoria (el mecanismo de esto se describe en el Apéndice A). Por tanto, tiene sentido que el método **hashCode( )** producido por dos instancias distintas de **new String("hola")** debería ser idéntico. Esto se puede comprobar ejecutando el siguiente programa:

```
//: c09:StringHashCode.java
```

```
public class StringHashCode {
  public static void main(String[] args) {
    System.out.println("Hola".hashCode());
    System.out.println("Hola".hashCode());
  }
} ///:~
```

Para que esto funcione, el método **hashCode( )** de **String** debe basarse en los contenidos de **String**.

Por tanto para que un método **hashCode( )** sea efectivo, debe ser rápido y significativo: es decir, debe generar un valor basado en los contenidos del objeto. Recuérdese que este valor no tiene por qué ser único —se debería preferir la velocidad más que la unicidad —pero entre **hashCode( )** y **equals( )** debería resolverse completamente la identidad del objeto.

Dado que **hashCode( )** se procesa antes de producir el índice de elementos, el rango de valores no es importante; simplemente es necesario generar un valor **entero**.

Hay aún otro factor: un buen método **hashCode( )** debería producir una distribución uniforme de los valores. Si los valores tienden a concentrarse en una zona, el objeto **HashMap** o el objeto **HashSet** estarán más cargados en algunas áreas y no serían tan rápidos como podrían ser con una función de hashing que produzca distribuciones uniformes.

He aquí un ejemplo que sigue estas líneas:

```
//: c09:CuentaString.java
// Creando un buen hashCode().
import java.util.*;

public class CuentaString {
  private String s;
  private int id = 0;
  private static ArrayList creado =
    new ArrayList();
  public CuentaString(String str) {
    s = str;
    creado.add(s);
    Iterator it = creado.iterator();
    // Id es el número total de instancias
    // de este string en uso por CuentaString:
    while(it.hasNext())
      if(it.next().equals(s))
        id++;
  }
  public String toString() {
    return "String: " + s + " id: " + id +
      " hashCode(): " + hashCode() + "\n";
  }
```

```
  public int hashCode() {
    return s.hashCode() * id;
  }
  public boolean equals(Object o) {
    return (o instanceof CuentaString)
      && s.equals(((CuentaString)o).s)
      && id == ((CuentaString)o).id;
  }
  public static void main(String[] args) {
    HashMap m = new HashMap();
    CuentaString[] cs = new CuentaString[10];
    for(int i = 0; i < cs.length; i++) {
      cs[i] = new CuentaString("hi");
      m.put(cs[i], new Integer(i));
    }
    System.out.println(m);
    for(int i = 0; i < cs.length; i++) {
      System.out.print("Buscando " + cs[i]);
      System.out.println(m.get(cs[i]));
    }
  }
} ///:~
```

CuentaString incluye un **String** y un **id** que representa el número de objetos **CuentaString** que contienen un **String** idéntico. El conteo lo lleva a cabo el constructor iterando a través del método estatico **ArrayList** donde se almacenan todos los **String**s.

Tanto **hashCode( )** como **equals( )** producen resultados basados en ambos campos; si se basara simplemente en un **String** o en el **id** habría coincidencias duplicadas para valores distintos.

Fíjese lo simple que es **hashCode( )**: el **hashCode( )** del **String** se multiplica por el **id**. Cuanto más pequeño sea **hashCode( )**, mejor (y más rápido).

En el método **main( )** se crea un conjunto de objetos **CuentaString**, utilizando el mismo **String** para mostrar que los duplicados crean valores únicos debido a **id**. Se muestra el **HashMap,** de forma que se puede ver cómo se almacena internamente (en un orden imperceptible) y se busca cada clave individualmente para demostrar que el mecanismo de búsqueda funciona correctamente.

Guardar referencias

La biblioteca **java.lang.ref** contiene un conjunto de clases que permiten mayor flexibilidad en la recolección de basura, siendo especialmente útiles cuando se tiene objetos grandes que pueden agotar la memoria. Hay tres clases heredadas de la clase abstracta **Reference**: **SoftReference**, **WeakReference**, y **PhantomReference**. Cada una proporciona un nivel de direccionamiento diferente por parte del recolector de basura, si el objeto en cuestión *sólo* es alcanzable a través de uno de estos objetos **Reference**.

El que un objeto sea *accesible* significa que en algún sitio del programa se puede encontrar el objeto. Esto podría significar que se tiene una referencia en la pila que va directa al objeto, pero también se podría tener una referencia a un objeto que tiene una referencia al objeto en cuestión; podría haber muchos enlaces intermedios. Si un objeto es accesible, el recolector de basura no puede liberar el espacio que usa porque sigue en uso por parte del programa. Si no se puede acceder a un objeto, no hay forma de que el programa lo use, por lo que es seguro que el recolector lo eliminará.

Se usan objetos **Reference** cuando se desea seguir guardando una referencia a ese objeto —se desea ser capaz de acceder a ese objeto— pero también se quiere permitir al recolector de basura liberar ese objeto. Por consiguiente, se tiene alguna forma de usar el objeto, pero si se está cerca de una saturación de la memoria, se permite la recolección del objeto.

Esto se logra usando un objeto **Reference** que es un intermediario entre el programador y la referencia ordinaria, *y* no debe haber referencias ordinarias al objeto (las no envueltas dentro de objetos **Reference**). Si el recolector de basura descubre que se puede acceder a un objeto mediante una referencia ordinaria, no liberará ese objeto.

Si se ordenan **SoftReference**, **WeakReference**, y **PhantomReference**, cada uno es más "débil" que el anterior en lo que a nivel de "accesibilidad" se refiere. Las referencias blandas son para implementar cachés sensibles. Las referencias débiles son para implementar "correspondencias canónicas" —en las que las instancias de los objetos pueden usarse simultáneamente en múltiples lugares del programa, para ahorrar espacio de almacenamiento— que no evitan que sus claves (o valores) puedan ser reclamadas. Las referencias fantasma (*phantom*) permiten organizar acciones de limpieza antes de su muerte de forma más flexible que lo que permite el mecanismo de finalización de Java.

Con **SoftReference**s y **WeakReference**s se puede elegir si ubicarlas en una **ReferenceQueue** (el dispositivo usado para acciones de limpieza antes de su muerte), pero sólo se puede construir una **PhantomReference** en una **ReferenceQueue**. He aquí una demostración:

```
//: c09:Referencias.java
// Demuestra los objetos Referencia
import java.lang.ref.*;

class MuyGrande {
  static final int SZ = 10000;
  double[] d = new double[SZ];
  String ident;
  public MuyGrande(String id) { ident = id; }
  public String toString() { return ident; }
  public void finalize() {
    System.out.println("Finalizando " + ident);
  }
}

public class Referencias {
  static ReferenceQueue rq= new ReferenceQueue();
```

```
  public static void comprobarCola() {
    Object inq = rq.poll();
    if(inq != null)
      System.out.println("In cola: " +
        (MuyGrande)((Reference)inq).get());
  }
  public static void main(String[] args) {
    int tamanio = 10;
    // 0, elegir el tamaño a través de la línea de comandos:
    if(args.length > 0)
      tamanio = Integer.parseInt(args[0]);
    SoftReference[] sa =
      new SoftReference[tamanio];
    for(int i = 0; i < sa.length; i++) {
      sa[i] = new SoftReference(
        new MuyGrande("Blando " + i), rq);
      System.out.println("Recien creado: " +
        (MuyGrande)sa[i].get());
      comprobarCola();
    }
    WeakReference[] wa =
      new WeakReference[tamanio];
    for(int i = 0; i < wa.length; i++) {
      wa[i] = new WeakReference(
        new MuyGrande("debil " + i), rq);
      System.out.println("Recien creado: " +
        (MuyGrande)wa[i].get());
      comprobarCola();
    }
    SoftReference s = new SoftReference(
      new MuyGrande("Blando"));
    WeakReference w = new WeakReference(
      new MuyGrande("Debil"));
    System.gc();
    PhantomReference[] pa =
      new PhantomReference[tamanio];
    for(int i = 0; i < pa.length; i++) {
      pa[i] = new PhantomReference(
        new MuyGrande("Fantasma " + i), rq);
      System.out.println("Recien creado: " +
        (MuyGrande)pa[i].get());
      comprobarCola();
    }
  }
} ///:~
```

Cuando se ejecute este programa (se querrá encauzar la salida a través de una utilidad "más" de forma que se pueda ver la salida en varias páginas), se verá que se recolectan todos los objetos, incluso aunque se siga teniendo acceso a los mismos a través del objeto **Reference** (para conseguir la referencia al objeto actual, se usa **get( )**). También se verá que **ReferenceQueue** siempre produce un objeto **Reference** que contiene un objeto **null**. Para hacer uso de esto, se puede heredar de la clase **Reference** concreta en la que se esté interesado y añadir más métodos útiles al nuevo tipo **Reference**.

El objeto HasMap débil (**WeakHashMap**)

La biblioteca de contenedores tiene un **Mapa** especial para guardar referencias débiles: el **WeakHashMap**. Esta clase se diseña para facilitar la creación de correspondencias canónicas. En este tipo de correspondencias, se ahorra espacio de almacenamiento haciendo sólo una instancia de un valor particular. Cuando el programa necesita ese valor, busca el objeto existente en el mapa y lo usa (en vez de crearlo de la nada). La correspondencia puede hacer los valores como parte de esta inicialización, pero es más probable que los valores se hagan bajo demanda.

Dado que esta técnica permite ahorrar espacio de almacenamiento, es muy conveniente que el **WeakHashMap** permita al recolector de basura limpiar automáticamente las claves y valores. No se tiene que hacer nada especial a las claves y valores a ubicar en el **WeakHashMap**; éstos se envuelven automáticamente en **WeakReference**s por parte del mapa. El disparador para permitir la limpieza es que la clave deje de estar en uso, como aquí se demuestra:

```
//: c09:MapeoCanonico.java
// Demuestra WeakHashMap.
import java.util.*;
import java.lang.ref.*;

class Clave {
  String ident;
  public Clave(String id) { ident = id; }
  public String toString() { return ident; }
  public int hashCode() {
    return ident.hashCode();
  }
  public boolean equals(Object r) {
    return (r instanceof Clave)
      && ident.equals(((Clave)r).ident);
  }
  public void finalize() {
    System.out.println("Finalizando la Clave "+ ident);
  }
}

class Valor {
  String ident;
```

```
  public Valor(String id) { ident = id; }
  public String toString() { return ident; }
  public void finalize() {
    System.out.println("Finalizando el Valor "+ident);
  }
}

public class MapeoCanonico {
  public static void main(String[] args) {
    int tamanio = 1000;
    // O, elegir el tamaño mediante la línea de comandos:
    if(args.length > 0)
      tamanio = Integer.parseInt(args[0]);
    Clave[] claves = new Clave[tamanio];
    WeakHashMap whm = new WeakHashMap();
    for(int i = 0; i < size; i++) {
      Clave k = new Clave(Integer.toString(i));
      Valor v = new Valor(Integer.toString(i));
      if(i % 3 == 0)
        claves[i] = k; // Salvar como referencias "reales"
      whm.put(k, v);
    }
    System.gc();
  }
} ///:~
```

La clase **Clave** debe tener un método **hashCode( )** y un método **equals( )** dado que se está usando como clave en una estructura de datos con tratamiento hash, como se describió previamente en este capítulo.

Cuando se ejecute el programa se verá que el recolector de basura se saltará la tercera clave, pues también se ha ubicado esa clave en el array **claves** y, por consiguiente, estos objetos no podrán ser recolectados.

Revisitando los iteradores

Ahora se puede demostrar la verdadera potencia del **Iterador**: la habilidad de separar la operación de recorrer una secuencia de la estructura subyacente de esa secuencia. En el ejemplo siguiente, la clase **EscribirDatos** usa un **Iterador** para recorrer una secuencia y llama al método **toString( )** para cada objeto. Se crean dos tipos de contenedores distintos —un **ArrayList** y un **HashMap**— y se rellenan respectivamente con objetos **Ratón** y **Hamster**. (Estas clases se definen anteriormente en este capítulo.) Dado que un **Iterador** esconde la estructura del contenedor subyacente, **EscribirDatos** no sabe o no le importa de qué tipo de contenedor proviene el **Iterador**:

```
//: c09:Iteradores2.java
// Revisitando los Iteradores.
```

```
import java.util.*;

class EscribirDatos {
  static void escribir(Iterator e) {
    while(e.hasNext())
      System.out.println(e.next());
  }
}

class Iteradores2 {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 5; i++)
      v.add(new Raton(i));
    HashMap m = new HashMap();
    for(int i = 0; i < 5; i++)
      m.put(new Integer(i), new Hamster(i));
    System.out.println("ArrayList");
    EscribirDatos.print(v.iterator());
    System.out.println("HashMap");
    EscribirDatos.print(m.entrySet().iterator());
  }
} ///:~
```

Para el objeto **HashMap como**, el método **entrySet( )** produce un **conjunto** de objetos **Map.entry**, que contiene tanto la clave como el valor de cada entrada, por lo que se visualizan los dos.

Fíjese que **EscribirDatos.escribir( )** saca ventaja del hecho de que estos contenedores son de clase **Object**, por lo que la llamada a **toString( )** por parte de **System.out.println( )** es automática. Es más probable que en un problema haya que asumir que un **Iterador** esté recorriendo un contenedor de algún tipo específico. Por ejemplo, se podría imaginar que lo que contiene el contenedor es un **Polígono** con un método **dibujar( )**. Entonces habría que hacer una conversión hacia abajo del **Object** devuelto por **Iterator.next( )** para producir un **Polígono**.

Elegir una implementación

Hasta ahora, debería haberse entendido que sólo hay tres componentes contenedores: **Map**, **List** y **Set**, y sólo dos o tres implementaciones de cada interfaz. Si se necesita la funcionalidad ofrecida por una **interfaz** particular ¿cómo se decide qué implementación particular usar?

Para entender la respuesta, hay que ser consciente de que cada implementación diferente tiene sus propias caracteríticas, ventajas y desventajas. Por ejemplo, se puede ver en el diagrama que la "característica" de **Hashtable**, **Vector**, y **Stack** es que son clases antiguas, de forma que el código antiguo sigue funcionando. Por otro lado, es mejor si no se utilizan para código nuevo (Java 2).

La diferencia entre los otros contenedores suele centrarse en aquello por lo que están "respaldados"; es decir, las estructuras de datos que implementan físicamente la **interfaz** deseada. Esto significa que, por ejemplo, **ArrayList** y **LinkedList** implementan la interfaz **List**, de forma que el programa producirá los mismos resultados independientemente del que se use.

Sin embargo, un **ArrayList** está respaldado por un array, mientras que el **LinkedList** está implementado de la forma habitual de una lista doblemente enlazada, como objetos individuales cada uno de los cuales contiene datos junto con referencias a los elementos previo y siguiente de la lista. Debido a esto, si se desean hacer muchas inserciones y retiradas en la parte central de la lista, la selección apropiada es **LinkedList**. (**LinkedList** también tiene funcionalidad adicional establecida en **AbstractSequentialList**). Si no, suele ser más rápido un **ArrayList**.

Como otro ejemplo, se puede implementar un objeto **Set,** bien como un **TreeSet** o bien como **HashSet**. Un **TreeSet** está respaldado por un **TreeMap** y está diseñado para producir un conjunto ordenado. Sin embargo, si se va a tener cantidades mayores de datos en el **Set**, el rendimiento de las inserciones de **TreeSet** decaerá. Al escribir un programa que necesite un **Set**, debería elegirse **HashSet** por defecto, y cambiar a **TreeSet** cuando es más importante tener un conjunto constantemente ordenado.

Elegir entre **Listas**

La manera más convincente de ver las diferencias entre las implementaciones de **List** es con una prueba de rendimiento. El código siguiente establece una clase base interna que se usa como sistema de prueba, después crea un array de clases internas anónimas, una para cada prueba. El método **probar( )** llama a cada una de estas clases internas. Este enfoque te permite insertar y eliminar sencillamente nuevos tipos de pruebas.

```
//: c09:RendimientoListas.java
// Demuestra diferencias de rendimiento en Listas.
import java.util.*;
import com.bruceeckel.util.*;

public class RendimientoListas {
  private abstract static class RealizarPrueba {
    String nombre;
    int tamanio; // Cantidad de pruebas
    RealizarPrueba(String nombre, int tamanio) {
      this.nombre = nombre;
      this.tamanio = tamanio;
    }
    abstract void probar(List a, int reps);
  }
  private static RealizarPruebas[] Pruebas = {
    new RealizarPruebas("get", 300) {
      void probar(List a, int reps) {
```

```
        for(int i = 0; i < reps; i++) {
          for(int j = 0; j < a.tamanio(); j++)
            a.get(j);
        }
      }
    },
    new RealizarPrueba("iteracion", 300) {
      void probar(List a, int reps) {
        for(int i = 0; i < reps; i++) {
          Iterator it = a.iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
    new RealizarPrueba("insercion", 5000) {
      void probar(List a, int reps) {
        int mitad = a.size()/2;
        String s = "test";
        ListIterator it = a.listIterator(mitad);
        for(int i = 0; i < tamanio * 10; i++)
          it.add(s);
      }
    },
    new RealizarPrueba("eliminacion", 5000) {
      void probar(List a, int reps) {
        ListIterator it = a.listIterator(3);
        while(it.hasNext()) {
          it.next();
          it.remove();
        }
      }
    },
  };
  public static void probar(List a, int reps) {
    // Un truco para imprimir el nombre de la clase:
    System.out.println("Probando " +
      a.getClass().getName());
    for(int i = 0; i < pruebas.length; i++) {
      Colecciones2.rellenar(a,
        Colecciones2.paises.inicializar(),
        pruebas[i].tamanio);
      System.out.print(tamanio[i].nombre);
      long t1 = System.currentTimeMillis();
      pruebas[i].probar(a, reps);
```

```
        long t2 = System.currentTimeMillis();
        System.out.println(": " + (t2 - t1));
      }
    }
    public static void pruebaArray(int reps) {
      System.out.println("Probar array como lista");
      // En un array  sólo puede hacer las dos primeras pruebas:
      for(int i = 0; i < 2; i++) {
        String[] sa = new String[pruebas[i].tamanio];
        Arrays2.rellenarl(sa,
          Colecciones2.Paises.reset());
        List a = Arrays.asList(sa);
        System.out.print(pruebas[i].nombre);
        long t1 = System.currentTimeMillis();
        pruebas[i].probar(a, reps);
        long t2 = System.currentTimeMillis();
        System.out.println(": " + (t2 - t1));
      }
    }
    public static void main(String[] args) {
      int reps = 50000;
      // O, elegir el número de repeticiones
      // a través de la línea de comandos:
      if(args.length > 0)
        reps = Integer.parseInt(args[0]);
      System.out.println(reps + " repeticiones");
      pruebaArray(reps);
      prueba(new ArrayList(), reps);
      prueba(new LinkedList(), reps);
      prueba(new Vector(), reps);
    }
} ///:~
```

La clase interna **RealizarPrueba** es **abstracta**, para proporcionar una clase base para las pruebas específicas. Contiene un **String** que se imprime al empezar la prueba, un parámetro **tamanio** que se usará para contener la cantidad de elementos o repeticiones de prueba, un constructor para inicializar los campos, y un método **abstracto probar( )** que hace el trabajo. Todos los tipos de prueba se encuentran en el array **prueba**, que se inicializa con distintas clases internas anónimas heredadas de **RealizarPrueba**. Para añadir o eliminar probar, simplemente hay que añadir o eliminar una definición de clase interna del array, y todo ocurre automáticamente.

Para comparar el acceso a arrays con el acceso a contenedores (fundamentalmente contra **ArrayList**), se crea una prueba especial para arrays envolviéndolo como una **Lista** utilizando **Arrays.asList( )**. Fíjese que sólo se pueden hacer en esta clase las dos primeras pruebas, puesto que no se pueden insertar o eliminar elementos de un array.

La **Lista** que se pasa a **probar( )** se rellena primero con elementos, después se cronometran todos los pruebas del array **pruebas.** Los resultados variarán de una máquina a otra; se pretende que sólo den un orden de comparación de magnitudes entre el rendimiento de los distintos contenedores. He aquí un resultado de la ejecución:

Tipo	Obtener	Iteración	Insertar	Eliminar
array	1430	3850	No aplicable	No aplicable
ArrayList	3070	12200	500	46850
LinkedList	16320	9110	110	60
Vector	4890	16250	550	46850

Como se esperaba, los arrays son más rápidos que cualquier contenedor si se persigue el acceso aleatorio e iteraciones. Se puede ver que los accesos aleatorios (**get( )**) son baratos para objetos **ArrayList** y caros en el caso de objetos **LinkedList**. (La iteración es *más rápida* para una **LinkedList** que para un **ArrayList,** lo que resulta bastante intuitivo). Por otro lado, las inserciones y eliminaciones que se lleven a cabo en el medio de la lista son drásticamente más rápidas en una **LinkedList** que en una **ArrayList** —*especialmente* las eliminaciones. **Vector** es generalmente menos rápido que **ArrayList**, por lo que se recomienda evitarlo; sólo está en la biblioteca para dar soporte al código antiguo (la única razón por la que funciona en este programa es por que fue adaptada para ser una **Lista** en Java 2). El mejor enfoque es probablemente elegir un **ArrayList** por defecto, y cambiar a **LinkedList** si se descubren problemas de rendimiento debido a muchas inserciones y eliminaciones en el centro de la lista. Y por supuesto, si se está trabajando con un grupo de elementos de tamaño fijo, debe usarse un array.

Elegir entre **Conjuntos**

Se puede elegir entre un objeto **TreeSet** y un objeto **HashSet**, dependiendo del tamaño del **Conjunto** (si se necesita producir una secuencia ordenada de un **Conjunto**, se usará un **TreeSet**). El siguiente programa de pruebas da una indicación de este equilibrio:

```
//: c09:RendimientoConjuntos.java
import java.util.*;
import com.bruceeckel.util.*;

public class RendimientoConjuntos {
  private abstract static class RealizarPrueba {
    String nombre;
    RealizarPrueba(String nombre) { this.nombre = nombre; }
    abstract void probar(Set s, int tamanio, int reps);
  }
  private static RealizarPrueba[] pruebas = {
```

```
    new RealizarPrueba("insertar") {
      void probar(Set s, int tamanio, int reps) {
        for(int i = 0; i < reps; i++) {
          s.clear();
          Colecciones2.rellenar(s,
            Colecctiones2.paises.inicializar(),tamanio);
        }
      }
    },
    new RealizarPrueba("contiene") {
      void probar(Set s, int tamanio, int reps) {
        for(int i = 0; i < reps; i++)
          for(int j = 0; j < tamanio; j++)
            s.contains(Integer.toString(j));
      }
    },
    new RealizarPrueba("iteracion") {
      void probar(Set s, int tamanio, int reps) {
        for(int i = 0; i < reps * 10; i++) {
          Iterator it = s.iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
  };
  public static void
  probar(Set s, int tamanio, int reps) {
    System.out.println("Probando " +
      s.getClass().getName() + " tamanio " + tamanio);
    Colecciones2.rellenar1(s,
      Colecciones2.paises.inicializar(), tamanio);
    for(int i = 0; i < pruebas.length; i++) {
      System.out.print(probar[i].nombre);
      long t1 = System.currentTimeMillis();
      pruebas[i].probar(s, tamanio, reps);
      long t2 = System.currentTimeMillis();
      System.out.println(": " +
        ((double)(t2 - t1)/(double)tamanio));
    }
  }
  public static void main(String[] args) {
    int reps = 50000;
    // O, elegir el número de repeticiones
    // a través de la línea de comandos:
```

```
    if(args.length > 0)
      reps = Integer.parseInt(args[0]);
    // Pequeño:
    probar(new TreeSet(), 10, reps);
    probar(new HashSet(), 10, reps);
    // Medio:
    probar(new TreeSet(), 100, reps);
    probar(new HashSet(), 100, reps);
    // Grande:
    probar(new TreeSet(), 1000, reps);
    probar(new HashSet(), 1000, reps);
  }
} ///:~
```

Tipo	Tamaño prueba	Inserción	Contiene	Iteración
TreeSet	10	138,0	115,0	187,0
	100	189,5	151,1	206,5
	1000	150,6	177,4	40,04
HashSet	10	55,0	82,0	192,0
	100	45,6	90,0	202,2
	1000	36,14	106,5	39,39

La tabla siguiente muestra los resultados de la ejecución. (Por supuesto, éstos serán distintos en función del ordenador y la Máquina Virtual de Java que se esté usando; cada uno debería ejecutarlo):

El rendimiento de **HashSet** suele ser superior al de **TreeSet** en todas las operaciones (y especialmente en las dos operaciones más importantes: la inserción de elementos y la búsqueda). La única razón por la que existe **TreeSet** es porque mantiene sus elementos ordenados, por lo que se usa cuando se necesita un **Conjunto** ordenado.

Elegir entre **Mapas**

Al elegir entre implementaciones de **Mapas**, lo que más afecta al rendimiento es su tamaño; el siguiente programa de pruebas da una idea de este equilibrio:

```
//: c09:RendimientoMapas.java
// Demuestra diferencias de rendimiento en Mapas.
import java.util.*;
import com.bruceeckel.util.*;

public class RendimientoMapas {
```

```
  private abstract static class RealizarPrueba {
    String nombre;
    RealizarPrueba(String nombre) { this.nombre = nombre; }
    abstract void probar(Map m, int tamanio, int reps);
  }
  private static RealizarPrueba[] pruebas = {
    new RealizarPrueba("poner") {
      void probar(Map m, int tamanio, int reps) {
        for(int i = 0; i < reps; i++) {
          m.clear();
          Colecciones2.rellenar(m,
            Colecciones2.geografia.inicializar(), tamanio);
        }
      }
    },
    new RealizarPrueba("quitar") {
      void probar(Map m, int tamanio, int reps) {
        for(int i = 0; i < reps; i++)
          for(int j = 0; j < tamanio; j++)
            m.get(Integer.toString(j));
      }
    },
    new RealizarPrueba("iteracion") {
      void probar(Map m, int tamanio, int reps) {
        for(int i = 0; i < reps * 10; i++) {
          Iterator it = m.entrySet().iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
  };
  public static void
  probar(Map m, int tamanio, int reps) {
    System.out.println("Probando " +
      m.getClass().getName() + " tamanio " + tamanio);
    Colecciones2.rellenar(m,
      Colecciones2.geografia.inicializar(), size);
    for(int i = 0; i < pruebas.length; i++) {
      System.out.print(pruebas[i].nombre);
      long t1 = System.currentTimeMillis();
      pruebas[i].probar(m, tamanio, reps);
      long t2 = System.currentTimeMillis();
      System.out.println(": " +
        ((double)(t2 - t1)/(double)tamanio));
```

```
    }
  }
  public static void main(String[] args) {
    int reps = 50000;
    // O, elegir el número de repeticiones
    // a través de la línea de comandos:
    if(args.length > 0)
      reps = Integer.parseInt(args[0]);
    // pequeño:
    probar(new TreeMap(), 10, reps);
    probar(new HashMap(), 10, reps);
    probar(new Hashtable(), 10, reps);
    // Medio:
    probar(new TreeMap(), 100, reps);
    probar(new HashMap(), 100, reps);
    probar(new Hashtable(), 100, reps);
    // Grande:
    probar(new TreeMap(), 1000, reps);
    probar(new HashMap(), 1000, reps);
    probar(new Hashtable(), 1000, reps);
  }
} ///:~
```

Debido a que el factor es el tamaño del **mapa**, se verá que las pruebas de tiempos dividen el tiempo por el tamaño para normalizar cada medida. He aquí un conjunto de resultados. (Los de cada uno serán distintos.)

Tipo	Tamaño de la prueba	Poner	Quitar	Iteración
TreeMap	10	143,0	110,0	186,0
	100	201,1	188,4	280,1
	1000	222,8	205,2	40,7
HashMap	10	66,0	83,0	197,0
	100	80,7	135,7	278,5
	1000	48,2	105,7	41,4
HashTable	10	61,0	93,0	302,0
	100	90,6	143,3	329,0
	1000	54,1	110,95	47,3

Como se podía esperar, el rendimiento de **Hashtable** es equivalente al de **HashMap**. (También se puede ver que **HashMap** es generalmente un poco más rápida. Se pretende que **HashMap** reemplace a **Hashtable**.) El **TreeMap** es generalmente más lento que el **HashMap**, así que ¿por qué usarlo? Así se podría usar en vez de como un **Mapa**, como una forma de crear una lista ordenada. El comportamiento de un árbol es tal que siempre está en orden y no hay que ordenarlo de forma especial. Una vez que se rellena un **TreeMap** se puede invocar a **keySet( )** para conseguir una vista del **Conjunto** de las claves, después a **toArray( )** para producir un array de esas claves. Se puede usar el método **estático Arrays.binarySearch( )** (que se discutirá más tarde) para encontrar rápidamente objetos en el array ordenado. Por supuesto, sólo se haría esto si, por alguna razón, el comportamiento de **HashMap** fuera inaceptable, puesto que **HashMap** está diseñado para encontrar elementos rápidamente. También se puede crear fácilmente un **HashMap** a partir de un **TreeMap** con una única creación de objetos. Finalmente, cuando se usa un **Mapa** la primera elección debe ser **HashMap**, y sólo si se necesita un **Mapa** constantemente ordenado deberá usarse **TreeMap**.

Ordenar y buscar elementos en **Listas**

Las utilidades para llevar a cabo la ordenación y búsqueda de elementos en **Listas** tienen los mismos nombres y parámetros que las de ordenar arrays de objetos, pero son métodos **estáticos** de **Colecciones** en vez de **Arrays**. Aquí hay un ejemplo, modificado a partir de **BuscarEnArray.java**:

```
//: c09:OrdenarBuscarEnLista.java
// Ordenando y buscando Listas con 'Colecciones.'
import com.bruceeckel.util.*;
import java.util.*;

public class OrdenarBuscarEnLista {
  public static void main(String[] args) {
    List lista = new ArrayList();
    Colecciones2.rellenar(list,
      Colecciones2.capitales, 25);
    System.out.println(lista + "\n");
    Collections.shuffle(lista);
    System.out.println("Despues de desordenar: "+lista);
    Collections.sort(lista);
    System.out.println(lista + "\n");
    Object clave = lista.get(12);
    int indice =
      Collections.binarySearch(lista, clave);
    System.out.println("Posicion de " + clave +
      " es " + indice + ", lista.get(" +
      indice + ") = " + lista.get(indice));
    ComparadorAlfabetico comp =
      new ComparadorAlfabetico();
    Colecciones.sort(lista, comp);
```

```
    System.out.println(lista + "\n");
    clave = lista.get(12);
    indice =
      Collections.binarySearch(lista, clave, comp);
    System.out.println("Posicion de " + key +
      " es " + indice + ", lista.get(" +
      indice + ") = " + lista.get(indice));
  }
} ///:~
```

El uso de estos métodos es idéntico al de los **Arrays**, pero se está usando una **Lista** en vez de un array. Al igual que ocurre al buscar y ordenar en arrays, si se ordena usando un **Comparador**, hay que usar el **binarySearch( )** del propio **Comparador**.

Este programa también muestra el método **shuffle( )** de las **Colecciones**, que genera un orden aleatorio para una **Lista**.

Utilidades

Hay otras muchas utilidades en las clases de tipo **Colección**:

enumeration(Collection)	Devuelve una **Enumeración** de las antiguas para el parámetro.
max(Collection) **min(Collection)**	Devuelve el elemento máximo o mínimo del parámetro utilizando el método natural de comparación para los objetos de la **Colección**.
max(Collection, Comparator) **min(Collection, Comparator)**	Devuelve el elemento máximo o mínimo de la **Colección** utilizando el **Comparador**.
reverse( )	Invierte el orden de todos los elementos.
copy(List destino, List origen)	Copia los elementos de origen a destino.
fill(List lista, Object o)	Reemplaza todos los elementos de la lista con o.
nCopies(int n, Object o)	Devuelve una **Lista** inmutable de tamaño n cuyas referencias apuntan a o.

Fíjese que **min( )** y **max( )** trabajan con objetos **Colección**, en vez de **Listas**, por lo que no hay que preocuparse de si la **Colección** está o no ordenada. (Como se mencionó anteriormente, *hay* que ordenar una **Lista** utilizando el método **sort( )** o un array antes de llevar a cabo una **binarySearch ( )**.)

Hacer inmodificable una **Colección** o un **Mapa**

A menudo es conveniente crear una versión de sólo lectura de una **Colección** o un **Mapa**. La clase de tipo **Colección** permite hacer esto pasando el contenedor original a un método que devuelve una versión de sólo lectura. Hay cuatro variantes de este método, una para **Collection** (por si no se desea crear una **Colección** de un tipo más específico), **List**, **Set** y **Map**. Este ejemplo muestra la forma adecuada de construir versiones de sólo lectura de cada uno de ellos:

```
//: c09:SoloLectura.java
// Usando los métodos Collections.unmodifiable.
import java.util.*;
import com.bruceeckel.util.*;

public class SoloLectura{
  static Colecciones2.StringGenerador gen =
    Colecciones2.Paises;
  public static void main(String[] args) {
    Collection c = new ArrayList();
    Colecciones2.rellenar(c, gen, 25); // Insertar datos
    c = Collections.unmodifiableCollection(c);
    System.out.println(c); // La lectura es correcta
    c.add("uno"); // No se puede cambiar

    List a = new ArrayList();
    Colecciones2.rellenar(a, gen.reset(), 25);
    a = Collections.unmodifiableList(a);
    ListIterator lit = a.listIterator();
    System.out.println(lit.next()); // La lectura es correcta
    lit.add("uno"); // No se puede cambiar

    Set s = new HashSet();
    Colecciones2.rellenar(s, gen.inicializar(), 25);
    s = Collections.unmodifiableSet(s);
    System.out.println(s); // La lectura es correcta
    //! s.add("uno"); // No se puede cambiar

    Map m = new HashMap();
    Colecciones2.rellenar(m,
      Colecciones2.geografia, 25);
    m = Collections.unmodifiableMap(m);
    System.out.println(m); // La lectura es correcta
    //! m.put("Javier", "Hola!");
  }
} ///:~
```

En cada caso, hay que rellenar el contenedor con datos significativos *antes* de hacerlos de sólo lectura. Una vez cargado, el mejor enfoque es reemplazar la referencia existente por la producida por la llamada "inmodificable". De esa forma, no se corre el riesgo de cambiar accidentalmente los contenidos una vez convertida en inmodificable. Por otro lado, esta herramienta también permite mantener un contenedor modificado **privado** dentro de una clase, y devolver una referencia de sólo lectura a ese contenedor desde una llamada a un método. Por tanto se puede cambiar dentro de la clase, y el resto de gente simplemente puede leerlo.

Llamar al método "inmodificable" para un tipo particular no provoca comprobación en tiempo de ejecución, pero una vez que se ha dado la transformación, todas las llamadas a métodos que modifiquen los contenidos de un contenedor en particular producirán una excepción de tipo **UnsupportedOperationException**.

Sincronizar una **Colección** o **Mapa**

La palabra clave **synchronized** es una parte importante del *multihilo*, un tema aún más complicado en el que no se entrará hasta el Capítulo 14. Aquí, simplemente se destaca que la clase **Collections** contiene una forma de sincronizar automáticamente un contenedor entero. La sintaxis es similar a los métodos "inmodificable":

```
//: c09:Sincronizacion.java
// Uso de los métodos Collections.synchronized.
import java.util.*;

public class Sincronizacion {
  public static void main(String[] args) {
    Collection c =
      Collections.synchronizedCollection(
        new ArrayList());
    List list = Collections.synchronizedList(
      new ArrayList());
    Set s = Collections.synchronizedSet(
      new HashSet());
    Map m = Collections.synchronizedMap(
      new HashMap());
  }
} ///:~
```

En este caso, se pasa inmediatamente el contenedor nuevo a través del método "sincronizado" apropiado; de esa manera no hay forma de exponer accidentalmente la versión sin sincronizar.

Fallo rápido

Los contenedores de Java tienen también un mecanismo para evitar que los contenidos de un contenedor sean modificados por más de un proceso. El problema se da cuando se está iterando a través de un contenedor y algún proceso irrumpe insertando, retirando o cambiando algún objeto de

ese contenedor. Ese objeto puede estar aún por procesar o ya se ha pasado por él, o incluso puede ser que se den problemas al invocar a **size( )** —hay demasiados escenarios que conducen al desastre. La biblioteca de contenedores de Java incorpora un mecanismo de *fallo rápido* que vigila que no se den en el contenedor más cambios que aquéllos de los que cada proceso se responsabiliza. Si detecta que alguien más está modificando el contenedor, produce inmediatamente una excepción **ConcurrentModificationException**. Éste es el aspecto "*Fallo rápido*" —no intenta detectar un problema más tarde utilizando algoritmos más complejos.

Es bastante sencillo ver el mecanismo fallo rápido en operación —todo lo que se tiene que hacer es crear un iterador y añadir algo a la colección apuntada por el iterador, como en:

```
//: c09:FalloRapido.java
// Demuestra el comportamiento "fallo rápido".
import java.util.*;

public class FalloRapido {
  public static void main(String[] args) {
    Collection c = new ArrayList();
    Iterator it = c.iterator();
    c.add("Un objeto");
    // Origina una excepción:
    String s = (String)it.next();
  }
} ///:~
```

Se da una excepción porque se ha colocado algo en el contenedor *después* de que se adquiere un iterador para el contenedor. La posibilidad de que dos partes del programa puedan estar modificando el mismo contenedor produce un estado incierto, por lo que la excepción notifica que habría que cambiar el código —en este caso, hay que adquirir el iterador *después* de que se hayan añadido todos los elementos del contenedor.

Fíjese que nadie se puede beneficiar de este tipo de monitorización cuando se esté accediendo a los elementos de una **Lista** utilizando **get( )**.

Operaciones no soportadas

Es posible convertir un array en una **Lista** con el método **Arrays.asList( )**:

```
//: c09:NoSoportable.java
// ¡En ocasiones los métodos definidos en las
// interfaces Collection no funcionan!
import java.util.*;

public class NoSoportable {
  private static String[] s = {
    "uno", "dos", "tres", "cuatro", "cinco",
```

```
    "siete", "ocho", "nueve", "diez",
  };
  static List a = Arrays.asList(s);
  static List a2 = a.subList(3, 6);
  public static void main(String[] args) {
    System.out.println(a);
    System.out.println(a2);
    System.out.println(
      "a.contains(" + s[0] + ") = " +
      a.contains(s[0]));
    System.out.println(
      "a.containsAll(a2) = " +
      a.containsAll(a2));
    System.out.println("a.isEmpty() = " +
      a.isEmpty());
    System.out.println(
      "a.indexOf(" + s[5] + ") = " +
      a.indexOf(s[5]));
    // Recorrer hacia atrás:
    ListIterator lit = a.listIterator(a.size());
    while(lit.hasPrevious())
      System.out.print(lit.previous() + " ");
    System.out.println();
    // Poner distintos elementos a los valores:
    for(int i = 0; i < a.size(); i++)
      a.set(i, "47");
    System.out.println(a);
    // Compila, pero no se ejecuta:
    lit.add("X"); // Operación no soportada
    a.clear(); // No soportada
    a.add("once"); // No soportada
    a.addAll(a2); // No soportada
    a.retainAll(a2); // No soportada
    a.remove(s[0]); // No soportada
    a.removeAll(a2); // No soportada
  }
} ///:~
```

Se descubrirá que sólo están implementados algunas de las interfaces de **List** y **Collection**. El resto de los métodos causan la aparición no bienvenida de algo denominado **UnsupportedOperationException**. Se aprenderá todo lo relativo a las excepciones en el capítulo siguiente, pero en resumidas cuentas, la **interfaz Collection** —al igual que algunos de las demás interfaces de la biblioteca de contenedores de Java— contienen métodos "opcionales", que podrían estar o no "soportados" en la clase concreta que **implementa** esa **interfaz**. Llamar a un método no

soportado causa una **UnsupportedOperationException** para indicar un error de programación.

"¡Qué!" dice uno incrédulo. "¡La razón principal de las **interfaces** y las clases base es que prometen que estos métodos harán algo significativo! Esto rompe esa promesa —dice que los métodos invocados *no* sólo *no* desempeñarán un comportamiento significativo, sino que pararán el programa! ¡Nos acabamos de cargar la seguridad a nivel de tipos!"

No es para tanto. Con un objeto de tipo **Collection**, **Set**, **List** o **Map**, el compilador sigue restringiendo de forma que sólo se invoquen los métodos de esta **interfaz**, a diferencia de lo que ocurre en Smalltalk (en el que se puede invocar a cualquier método para cualquier objeto, no descubriendo hasta tiempo de ejecución que la llamada no hace nada). Además, la mayoría de los métodos que toman una **Colección** como un parámetro sólo leen de la misma —y *ninguno* de los métodos de "lectura" de las **Colecciones** es opcional.

Este enfoque evita una explosión de interfaces en el diseño. Otros diseños para las bibliotecas de contenedores siempre parecen acabar con una plétora de interfaces para describir cada una de las variaciones, siendo por consiguiente, difíciles de aprender. Ni siquiera es posible capturar todos los casos especiales en **interfaces**, porque siempre habrá alguien capaz de inventar una **interfaz** nueva. El enfoque de "operación no soportada" logra una meta importante de la biblioteca de contenedores de Java: los contenedores se vuelven fáciles de aprender y usar; las operaciones no soportadas son un caso especial que puede aprenderse más adelante. Sin embargo, para que este enfoque funcione:

1. La **UnsupportedOperationException** debe ser un evento extraño. Es decir, en la mayoría de clases deberían funcionar todas las operaciones, y sólo en casos especiales podría haber una operación sin soporte. Esto es cierto en la biblioteca de contenedores de Java, dado que las clases que se usan el 99 % de las veces —**ArrayList**, **LinkedList**, **HashSet** y **HashMap**, al igual que las otras implementaciones concretas— soportan todas las operaciones. El diseño sí que proporciona una "puerta trasera" si se desea crear una nueva **Colección** sin proporcionar definiciones significativas para todos los métodos de la Interfaz **Collection**, haciendo, sin embargo, que encaje en la biblioteca existente.

2. Cuando una operación *no* tiene soporte, debería haber una probabilidad razonable de que aparezca una **UnsupportedOperationException** en tiempo de implementación, más que cuando se haya entregado el producto al cliente. Después de todo, indica un error de programación: se ha usado una implementación de forma incorrecta. Este punto es menos detectable, y es donde se pone en juego la naturaleza experimental del diseño. Sólo el tiempo permitirá ir averiguando con certeza cómo funciona.

En el ejemplo de arriba, **Arrays.asList( )** produce una **Lista** soportada por un array de tamaño fijo. Por consiguiente, tiene sentido que sólo sean las operaciones soportadas las que no cambien el tamaño del array. Si, por otro lado, se requiriera una nueva **interfaz** para expresar este tipo de comportamiento distinto (denominado, quizás, "**FixedSizeList**"), conduciría directamente a la complejidad y pronto se dejaría de saber dónde empezar a intentar usar la biblioteca.

La documentación para un método que tome una **Collection**, **List**, **Set** o **Map** como parámetro debería especificar cuál de los métodos opcionales debe implementarse. Por ejemplo, la ordenación requiere métodos **set( )** e **Iterator.set( )**, pero no **add( )** y **remove( )**.

Contenedores de Java 1.0/1.1

Desgraciadamente, se ha escrito mucho código utilizando los contenedores de Java 1.0/1.1, e incluso se escribe código nuevo utilizando estas clases. Por tanto, aunque nunca se debería utilizar nuevo código utilizando los contenedores viejos, hay que ser consciente de que existen. Sin embargo, los contenedores viejos eran bastante limitados, por lo que no hay mucho que decir de los mismos. (Puesto que son cosas del pasado, intentaremos evitar poner demasiado énfasis en algunas horribles decisiones de diseño.)

Vector y enumeration

La única secuencia cuyo tamaño podía autoexpandirse en Java 1.0/1.1 era el **Vector**, y por tanto se usó mucho. Tiene demasiados defectos como para describirlos aquí (véase la primera edición de este libro, disponible en el CD ROM de este libro y como descarga gratuita de: *http://www.Bruce Eckel.com*). Básicamente, se puede pensar que es un **ArrayList** con nombres de métodos largos y extraños. En la biblioteca de contenedores de Java 2 se ha adaptado **Vector,** de forma que pudiera encajar como una **Colección** y una **Lista**, por lo que en el ejemplo siguiente, el método **Colecciones2.rellenar( )** se usa exitosamente. Esto resulta un poco extraño, pues puede confundir a la gente que puede llegar a pensar que **Vector** ha mejorado, cuando, de hecho, sólo se ha incluido para soportar el código previo a Java 2.

La versión Java 1.0/1.1 del iterador decidió inventar un nuevo nombre: "enumeración", en vez de utilizar un término con el que todo el mundo ya era familiar. La interfaz **Enumeration** es más pequeño que **Iterator**, con sólo dos métodos, y usa nombres de método más largos: **boolean hasMoreElements( )** devuelve **verdadero** si esta enumeración contiene más elementos, y **Object nextElement( )** devuelve el siguiente elemento de la enumeración actual si es que hay alguno (de otra manera, produce una excepción).

Enumeration es sólo una interfaz, no una implementación, e incluso las bibliotecas nuevas siguen usando la vieja **Enumeration** —que es desdichada, pero generalmente inocua. Incluso aunque se debería usar siempre **Iterador** cuando se pueda, hay que estar preparados para bibliotecas que quieran hacer uso de una **Enumeración**.

Además, se puede producir una **Enumeración** para cualquier **Colección** utilizando el método **Collections.enumeration( )**, como se ve en este ejemplo:

```
//: c09:Enumeraciones.java
// Java 1.0/1.1 Vector y Enumeration.
import java.util.*;
import com.bruceeckel.util.*;
```

```
class Enumeraciones {
  public static void main(String[] args) {
    Vector v = new Vector();
    Colecciones2.rellenar(
      v, Colecciones2.paises, 100);
    Enumeration e = v.elements();
    while(e.hasMoreElements())
      System.out.println(e.nextElement());
    // devuelve una enumeración de una colección:
    e = Collections.enumeration(new ArrayList());
  }
} ///:~
```

El **Vector** de Java 1.0/1.1 sólo tiene un método **addElement( )**, pero **rellenar( )** hace uso del método **add( )** que se adaptó cuando se convirtió **Vector** en **Lista**. Para producir una **Enumeración**, se invoca a **elements( )**, por lo que se puede usar para llevar a cabo una iteración hacia adelante.

La última línea crea un **ArrayList** y usa **enumeration( )** para adaptar una **Enumeración** del **iterador** de **ArrayList**. Por consiguiente, si se tiene código viejo que requiera una **Enumeración**, se pueden seguir usando los nuevos contenedores.

Hashtable

Como se ha visto en la comparación de rendimientos en este capítulo, la **Hashtable** básica es muy similar al **HashMap**, incluso en los nombres de método. No hay razón para usar **Hashtable** en vez de **HashMap** en el código nuevo.

Pila (Stack)

El concepto de pila ya se presentó anteriormente, con la **LinkedList**. Lo extraño de la **Pila** de Java 1.0/1.1 es que en vez de usar **Vector** como bloque constructivo, la **Pila** se *hereda* de **Vector**. Por tanto tiene todas las características y comportamientos de un **Vector** más algunos comportamientos propios de **Pila**. Es difícil saber si los diseñadores decidieron explícitamente que ésta fuera una forma especialmente útil de hacer las cosas, o si fue simplemente un diseño ingenuo.

He aquí una simple demostración de una **Pila** que introduce cada línea de un array de **Cadenas de caracteres**:

```
//: c09:Pilas.java
// Demostración de la clase Stack.
import java.util.*;

public class Pilas {
  static String[] meses = {
    "Enero", "Febrero", "Marzo", "Abril",
```

```
      "Mayo", "Junio", "Julio", "Agosto", "Septiembre",
      "Octubre", "Noviembre", "Diciembre" };
   public static void main(String[] args) {
      Stack pila = new Stack();
      for(int i = 0; i < meses.length; i++)
        pila.push(meses[i] + " ");
      System.out.println("pila = " + pila);
      // Tratando una pila como un Vector:
      pila.addElement("La ultima linea");
      System.out.println(
        "elemento 5 = " + pila.elementAt(5));
      System.out.println("sacando elementos:");
      while(!pila.empty())
        System.out.println(pila.pop());
   }
} ///:~
```

Cada línea del array **meses** se inserta en la **Pila** con **push( )** y posteriormente se la toma de la cima de la pila con **pop( )**. Todas las operaciones **Vector** también se ejecutan en el objeto **Stack**. Esto es posible porque, gracias a la herencia, un objeto de tipo **Stack** *es* un **Vector**. Por consiguiente, todas las operaciones que puedan llevarse a cabo sobre un **Vector** también pueden ejecutarse en una **Pila** como **elementAt( )**.

Como se mencionó anteriormente, se debería usar un objeto **LinkedList** cuando se desee comportamiento de pila.

Conjunto de bits (BitSet)

Un **Conjunto de bits** se usa si se desea almacenar eficientemente gran cantidad de información. Es eficiente sólo desde el punto de vista del tamaño; si se busca acceso eficiente, es ligeramente más lento que usar un array de algún tipo nativo.

Además, el tamaño mínimo de **Conjunto de bits** es el de un **long**: 64 bits. Esto implica que si se está almacenando algo menor, como 8 bits, un **Conjunto de bits** sería un derroche; es mejor crear una clase o simplemente un array, para guardar indicadores cuando el tamaño es importante.

Un contenedor normal se expande al añadir más elementos, y un **Conjunto de bits** también. El ejemplo siguiente muestra el funcionamiento del **Conjunto de bits**:.

```
//: c09:Bits.java
// Demostración de BitSet.
import java.util.*;

public class Bits {
  static void escribirBitset(BitSet b) {
    System.out.println("bits: " + b);
```

```
    String bbits = new String();
    for(int j = 0; j < b.size() ; j++)
      bbits += (b.get(j) ? "1" : "0");
    System.out.println("patron de bit: " + bbits);
  }
  public static void main(String[] args) {
    Random aleatorio = new Random();
    // Toma el LSB de nextInt():
    byte bt = (byte)aleatorio.nextInt();
    BitSet bb = new BitSet();
    for(int i = 7; i >=0; i--)
      if(((1 << i) &  bt) != 0)
        bb.set(i);
      else
        bb.clear(i);
    System.out.println("valor byte: " + bt);
    escribirBitSet(bb);

    short st = (short)aleatorio.nextInt();
    BitSet bs = new BitSet();
    for(int i = 15; i >=0; i--)
      if(((1 << i) &  st) != 0)
        bs.set(i);
      else
        bs.clear(i);
    System.out.println("valor short: " + st);
    escribirBitSet(bs);

    int it = aleatorio.nextInt();
    BitSet bi = new BitSet();
    for(int i = 31; i >=0; i--)
      if(((1 << i) &  it) != 0)
        bi.set(i);
      else
        bi.clear(i);
    System.out.println("valor int: " + it);
    escribirBitSet(bi);

    // Prueba conjunto de bits >= 64 bits:
    BitSet b127 = new BitSet();
    b127.set(127);
    System.out.println("poner a uno el bit 127: " + b127);
    BitSet b255 = new BitSet(65);
    b255.set(255);
    System.out.println("poner a uno el bit 255: " + b255);
```

```
    BitSet b1023 = new BitSet(512);
    b1023.set(1023);
    b1023.set(1024);
    System.out.println("poner a uno el bit 1023: " + b1023);
  }
} ///:~
```

El generador de números aleatorios se usa para crear un **byte**, **short**, e **int** al azar, transformando cada uno en el patrón de bits correspondiente en el **Conjunto de bits**. Esto funciona bien porque un **Conjunto de bits** tiene 64 bits, por lo que ninguno de éstos provoca un incremento de tamaño. Posteriormente se crea un **Conjunto de bits** de 512 bits. El constructor asigna espacio de almacenamiento para el doble de bits. Sin embargo, se sigue pudiendo poner a uno el bit 1.024 o mayor.

Resumen

Para repasar los contenedores proporcionados por la biblioteca estándar de Java:

1. Un array asocia índices numéricos a objetos. Guarda objetos de un tipo conocido por lo que no hay que convertir el resultado cuando se está buscando un objeto. Puede ser multidimensional, y puede guardar datos primitivos. Sin embargo, no se puede cambiar su tamaño una vez creado.

2. Una **Colección** guarda elementos sencillos, mientras que un **Mapa** guarda pares asociados.

3. Como un array, una **Lista** también asocia índices numéricos a los objetos —se podría pensar que los arrays y **Lista**s son contenedores ordenados. La **Lista** se redimensiona automáticamente al añadir más elementos. Pero una **Lista** sólo puede guardar **referencias a Objetos**, por lo que no guardará datos primitivos, y siempre hay que convertir el resultado para extraer una referencia a un **Objeto** fuera del contenedor.

4. Utilice un **ArrayList** si se están haciendo muchos accesos al azar, y una **LinkedList** si se están haciendo muchas inserciones y eliminaciones en el medio de la lista.

5. El comportamiento de las colas, bicolas y pilas se proporciona mediante **LinkedList**.

6. Un **Mapa** es una forma de asociar valores no números, sino *objetos* con otros objetos. El diseño de un **HashMap** se centra en el acceso rápido, mientras que un **TreeMap** mantiene sus claves ordenadas, no siendo, por tanto, tan rápido como un **HashMap**.

7. Un **Conjunto** sólo acepta uno de cada tipo de objeto. Los **HashSets** proporcionan búsquedas extremadamente rápidas, mientras que los **TreeSet**s mantienen los elementos ordenados.

8. No hay necesidad de usar las clases antiguas **Vector**, **Hashtable** y **Stack** en el código nuevo.

Los contenedores son herramientas que se pueden usar en el día a día para construir programas más simples, más potentes y más efectivos.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com*.

1. Crear un array de **elementos de tipo doble** y rellenarlo haciendo uso de **GeneradorDobleAleatorio**. Imprimir los resultados.
2. Crear una nueva clase denominada **Jerbo** con un **atributo numeroJerbos de tipo entero** que se inicialice en el constructor (similar al ejemplo **Ratón** de este capítulo). Darle un método denominado **saltar( )** que imprima qué numero de jerbo se está saltando. Crear un **ArrayList** y añadir un conjunto de objetos **Jerbo** a la **Lista**. Ahora usar el método **get( )** para recorrer la **Lista** y llamar a **saltar( )** para cada **Jerbo**.
3. Modificar el Ejercicio 2 de forma que se use un **Iterator** para recorrer la **Lista** mientras se llama a **saltar( )**.
4. Coger la clase **Jerbo** del Ejercicio 2 y ponerla en un **Mapa** en su lugar, asociando el nombre del **Jerbo** como un **String** (la clave) por cada **Jerbo** (el valor) que se introduzca en la tabla. Conseguir un **Iterador** para el **keySet( )** y utilizarlo para recorrer **Mapa**, buscando el **Jerbo** para cada clave e imprimiendo la clave y ordenando al **Jerbo** que **saltar( )**.
5. Crear una **Lista** (intentarlo tanto con **ArrayList** como con **LinkedList**) y rellenarla con **Colecciones2.paises**. Ordenar la lista e imprimirla, después aplicar **Collections.shuffle( )** a la lista repetidamente, imprimiéndola cada vez para ver como el método **shuffle( )** genera una lista aleatoriamente distinta cada vez.
6. Demostrar que no se puede añadir nada que no sea un **Ratón** a una **ListaRaton**.
7. Modificar **ListaRaton.java** de forma que herede de **ArrayList** en vez de hacer uso de la composición. Demostrar el problema con este enfoque.
8. Modificar **GatosYPerros.java** creando un contenedor **Gatos** (utilizando **ArrayList**) que sólo aceptará y retirará objetos **Gato**.
9. Crear un contenedor que encapsule un array de cadena de caracteres, y que sólo añada y extraiga cadena de caracteres, de forma que no haya problemas de conversiones durante su uso. Si el array interno no es lo suficientemente grande para la siguiente adición, redimensionar automáticamente el contenedor. En el método **main( )** comparar el rendimiento del contenedor con un **ArrayList** que almacene cadena de caracteres.
10. Repetir el Ejercicio 9 para un contenedor de **enteros**, y comparar el rendimiento con el de un **ArrayList** que guarde objetos **Integer**. En la comparación de rendimiento, incluir el proceso de incrementar cada objeto del contenedor.
11. Utilizando las utilidades de **com.bruceeckel.util**, crear un array de cada tipo primitivo y de **Cadena de Caracteres**, rellenar después cada array utilizando un generador apropiado, e imprimir cada array usando el método **escribir( )** adecuado.

12. Crear un generador que produzca nombres de personajes de alguna película (se puede usar por defecto *Blanca Nieves* o *La Guerra de las galaxias*), y que vuelva a comenzar por el principio al quedarse sin nombres. Utilizar las utilidades de **com.bruceeckel.util** para rellenar un array, un **ArrayList**, una **LinkedList** y ambos tipos de **Set**, para finalmente imprimir cada contenedor.

13. Crear una clase que contenga dos objetos **String**, y hacerla **Comparable,** de forma que la comparación sólo se encargue del primer **String**. Rellenar un array y un **ArrayList** con objetos de la clase, utilizando el generador **geografía**. Demostrar que la ordenación funciona correctamente. Hacer ahora un **Comparador** que sólo se encargue del segundo **String** y demostrar que la ordenación funciona correctamente; llevar a cabo también una búsqueda binaria haciendo uso del **Comparador**.

14. Modificar el Ejercicio 13 de forma que se siga un orden alfabético.

15. Utilizar **Arrays2.GeneradorStringAleatorio** para rellenar un **TreeSet** pero usando ordenación alfabética. Imprimir el **TreeSet** para verificar el orden.

16. Crear un **ArrayLit** y una **LinkedList**, y rellenar cada uno utilizando el generador **Collections2.capitales**. Imprimir cada lista utilizando un **Iterador** ordinario, y después insertar una lista dentro de la otra utilizando un **ListIterator**, intercalándolas. Llevar a cabo ahora la inserción empezando al final de la primera lista y recorriéndola hacia atrás.

17. Escribir un método que use un **Iterador** para recorrer una **Colección** e imprimir el **hashCode( )** de cada objeto del contenedor. Rellenar todos los tipos distintos de **Colección** con objetos y aplicar el método a cada contenedor.

18. Reparar el problema de **RecursividadInfinita.java**.

19. Crear una clase, después hacer un array inicializado de objetos de esa clase. Rellenar una **Lista** con ese array. Crear un subconjunto de la **Lista** utilizando **subList( )**, y eliminar después este subconjunto de la **Lista** utilizando **removeAll( )**.

20. Cambiar el Ejercicio 6 del Capítulo 7, de forma que use un **ArrayList** para guardar los **Roedores** y un **Iterador** para recorrer la secuencia de objctos **Roedores**. Recordar que un **ArrayList** guarda sólo **Objetos** por lo que hay que hacer una conversión al acceder a los objetos **Roedor** individuales.

21. Siguiendo el ejemplo **Cola.java**, crear una clase **Bicola** y probarla.

22. Utilizar un **TreeMap** en **Estadisticas.java**. Añadir ahora código que pruebe la diferencia de rendimiento entre **HashMap** y **TreeMap** en el programa.

23. Producir un **Mapa** y un **Conjunto** que contengan todos los países que empiecen por "A".

24. Utilizando **Colecciones2.paises**, rellenar un **Set** varias veces con los mismos datos y verificar que el **Set** acaba con sólo una instancia de cada. Intentarlo con los dos tipos de **Set**.

25. A partir de **Estadisticas.java**, crear un programa que ejecute la prueba repetidamente y comprueba si alguno de los números tiende a aparecer en los resultados más a menudo que otros.

26. Volver a escribir **Estadisticas.java** utilizando un **HashSet** de objetos **Contador** (habrá que modificar **Contador** de forma que trabaje en el **HashSet**). ¿Qué enfoque parece mejor?

27. Modificar la clase del Ejercicio 13 de forma que trabaje con objetos **HashSet**, y utilizar una clave de objeto **HashMap**.

28. Utilizando como inspiración **MapaLento.java**, crear un **MapaLento**.

29. Aplicar las pruebas de **Mapas1.java** a **MapaLento** para verificar que funciona. Arreglar todo lo que no funcione correctamente en **MapaLento**.

30. Implementar el resto de la interfaz **Map** para **MapaLento**.

31. Modificar **RendimientoMapa.java** para que incluya tests de **MapaLento**.

32. Modificar **MapaLento** para que en vez de objetos **ArrayList** guarde un único **ArrayList** de objetos **MPar**. Verificar que la versión modificada funcione correctamente. Utilizando **RendimientoMapa.java**, probar la velocidad del nuevo **Mapa**. Ahora cambiar el método **put( )** de forma que haga un **sort( )** después de que se introduzca cada par, y modificar **get( )** para que use **Collections.binarySearch( )** para buscar la clave. Comparar el rendimiento de la nueva versión con el de la vieja.

33. Añadir un campo **de tipo carácter** a **CuentaString** que se inicialice también en el constructor, y modificar los métodos **HashCode( )** y **equals( )** para incluir el valor de este **de tipo carácter**.

34. Modificar **SimpleHashMap** de forma que informe sobre colisiones, y probarlo añadiendo el mismo conjunto de datos dos veces, de forma que se observen colisiones.

35. Modificar **SimpleHashMap** de forma que informe del número de "intentos" necesarios cuando se dan colisiones. Es decir, ¿cuántas llamadas a **next( )** hay que hacer en los **Iteradores** que recorren las **LinkedLists** para encontrar coincidencias?

36. Implementar los métodos **clear( )** y **remove( )** para **SimpleHashMap**.

37. Implementar el resto de la interfaz **Map** para **SimpleHashMap**.

38. Añadir un método **privado rehash( )** para **SimpleHashMap** al que se invoca cuando el factor de carga excede 0,75. Durante el rehashing doblar el número de posiciones, después buscar el primer número primo mayor para determinar el nuevo número de posiciones.

39. Siguiendo el ejemplo de **SimpleHashMap.java**, crear y probar un **SimpleHashSet**.

40. Modificar **SimpleHashMap** para que use **ArrayList** en vez de **LinkedList**. Modificar **RendimientoMapa.java** para comparar el rendimiento de ambas implementaciones.

41. Utilizando la documentación HTML del JDK (descargable de *http://java.sun.com*), buscar la clase **HashMap**. Crear un **HashMap**, rellenarlo con elementos y determinar el factor de carga. Probar la velocidad de búsqueda con este mapa, y después intentar incrementar la velocidad haciendo un nuevo **HashMap** con una capacidad inicial más grande y copiando el mapa viejo en el nuevo, ejecutando de nuevo la prueba de velocidad de búsqueda en el nuevo mapa.

42. En el Capítulo 8, localizar el ejemplo **ControlesInvernadero.java**, que consta de tres ficheros. En **Controlador.java**, la clase **ConjuntoEventos** es simplemente un contenedor. Cambiar el código para usar una **LinkedList** en vez de un **ConjuntoEventos**. Esto exigirá más que simplemente reemplazar **ConjuntoEventos** con **LinkedList**; también se necesitará usar un **Iterador** para recorrer el conjunto de eventos.

43. (Desafío). Escribir una clase mapa propia con hashing, personalizada para un tipo de clave particular: **String** en este caso. No heredarla de **Map**. En su lugar, duplicar los métodos de forma que los métodos **put( )** y **get( )** tomen específicamente objetos **String**, en vez de **Objects**, como claves. Todo lo relacionado con las claves no debería usar tipos genéricos, sino que debería funcionar con **String**s, para evitar el coste de las conversiones hacia arriba y hacia abajo. La meta es hacer la implementación general más rápida posible. Modificar **RendimientoMapa.java** de forma que pruebe esta implementación contra un **HashMap**.

44. (Desafío). Encontrar el código fuente de **List** en la biblioteca de código fuente de Java que viene con todas las distribuciones de Java. Copiar este código y hacer una versión especial llamada **ListaEnteros** que guarde sólo **números enteros**. Considerar qué implicaría hacer una versión especial de **List** para todos los tipos primitivos. Considerar ahora qué ocurre si se desea hacer una clase lista enlazada que funcione con todos los tipos primitivos. Si alguna vez se llegaran a implementar tipos parametrizados en Java, proporcionarán la forma de hacer este trabajo automáticamente (además de muchos otros beneficios).