3.6 Clonación de estructuras de datos

La belleza de la programación orientada a objetos es que la abstracción permite que una estructura de datos sea tratada como un solo objeto, aunque la implementación encapsulada de la estructura pueda depender de una combinación más compleja de muchos objetos.
En esta sección, consideramos lo que significa hacer una copia de dicha estructura.

En un entorno de programación, se espera comúnmente que una copia de un objeto tenga su propio estado y que, una vez creada, sea independiente del original (por ejemplo, para que los cambios en uno no afecten directamente al otro). Sin embargo, cuando los objetos tienen campos que son variables de referencia que apuntan a objetos auxiliares, no siempre es evidente si una copia debe tener un campo correspondiente que haga referencia al mismo objeto auxiliar o a una nueva copia de dicho objeto.

Por ejemplo, si una clase hipotética AddressBook tiene instancias que representan una libreta de direcciones electrónica (con información de contacto, como números de teléfono y direcciones de correo electrónico, de los amigos y conocidos de una persona), ¿cómo podríamos imaginar una copia de la libreta de direcciones? ¿Debería aparecer una entrada añadida a una libreta en la otra? Si cambiamos el número de teléfono de una persona en un libro, ¿esperaríamos que ese cambio se sincronice en el otro?

No existe una respuesta universal para preguntas como esta. En cambio, cada clase en Java es responsable de definir si sus instancias se pueden copiar y, de ser así, cómo se construye la copia. La superclase universal Object define un método llamado clone, que permite generar una copia superficial de un objeto. Esta utiliza la semántica de asignación estándar para asignar el valor de cada campo del nuevo objeto al campo correspondiente del objeto existente que se está copiando. Esto se conoce como copia superficial porque, si el campo es un tipo de referencia, una inicialización con la forma duplicate.field = original.field hace que el campo del nuevo objeto haga referencia a la misma instancia subyacente que el campo del objeto original.

Una copia superficial no siempre es apropiada para todas las clases; por lo tanto, Java deshabilita intencionalmente el uso del método clone() al declararlo como protegido y al hacer que genere una excepción CloneNotSupportedException al ser llamado. El autor de una clase debe declarar explícitamente la compatibilidad con la clonación, declarando formalmente que la clase implementa la interfaz Cloneable y una versión pública del método clone(). Este método público puede simplemente llamar al método protegido para realizar la asignación campo por campo que resulta en una copia superficial, si corresponde. Sin embargo, para muchas clases, la clase puede optar por implementar una versión más avanzada de la clonación, en la que se clonan algunos de los objetos referenciados.

Para la mayoría de las estructuras de datos de este libro, omitimos la implementación de un método de clonación válido (dejándolo como ejercicio). Sin embargo, en esta sección, consideramos enfoques para clonar tanto arrays como listas enlazadas, incluyendo una implementación concreta del método de clonación para la clase SinglyLinkedList.

3.6.1 Clonación de matrices

Aunque las matrices admiten algunas sintaxis especiales como a[k] y a.length, es importante recordar que son objetos y que las variables de matriz son referencias.
Variables. Esto tiene consecuencias importantes. Como primer ejemplo, considere el siguiente código:

```
int[ ] datos = {2, 3, 5, 7, 11, 13, 17, 19};
int[ ] copia de seguridad;
copia de seguridad = datos;                    // advertencia; no es una copia
```

La asignación de una copia de seguridad variable a los datos no crea ninguna matriz nueva; simplemente crea un nuevo alias para la misma matriz, como se muestra en la Figura 3.23.

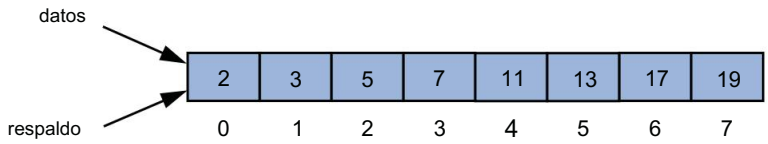

Figura 3.23: El resultado del comando backup = data para matrices int.

En cambio, si queremos hacer una copia de la matriz, datos y asignar una referencia a Para la nueva matriz a variable, copia de seguridad, debemos escribir:

```
copia de seguridad = datos.clone();
```

El método clone, cuando se ejecuta en una matriz, inicializa cada celda de la nueva matriz. al valor almacenado en la celda correspondiente de la matriz original. Esto da como resultado en una matriz independiente, como se muestra en la Figura 3.24.

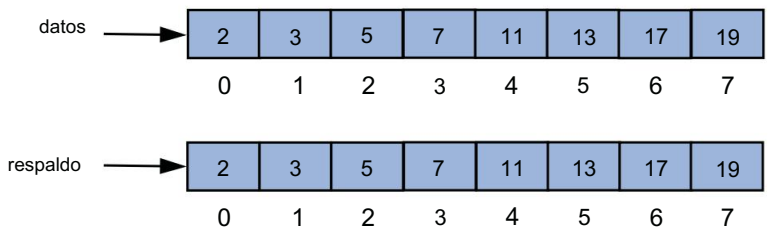

Figura 3.24: El resultado del comando backup = data.clone() para matrices int.

Si posteriormente realizamos una asignación como data[4] = 23 en esta configuración, La matriz de respaldo no se ve afectada.

Hay más consideraciones al copiar una matriz que almacena referencias tipos en lugar de tipos primitivos. El método clone() produce una copia superficial. de la matriz, produciendo una nueva matriz cuyas celdas hacen referencia a los mismos objetos referenciados por la primera matriz.

Por ejemplo, si la variable contactos se refiere a una matriz de personas hipotéticas
En algunas instancias, el resultado del comando guest = contacts.clone() produce una copia superficial, como se muestra en la Figura 3.25.

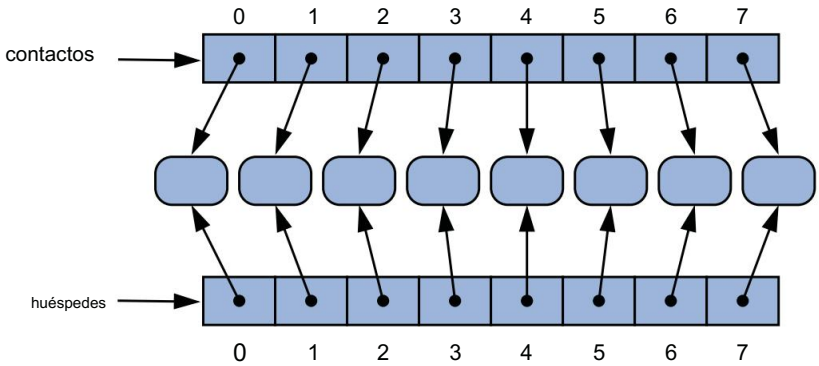

Figura 3.25: Una copia superficial de una matriz de objetos, resultante del comando invitados = contactos.clone( ).

Se puede crear una copia profunda de la lista de contactos clonando iterativamente los elementos individuales, de la siguiente manera, pero solo si la clase Persona se declara como Clonable.

```
Persona[ ] invitados = new Persona[contactos.length];
para (int k=0; k < contactos.longitud; k++)
   invitados[k] = (Persona) contactos[k].clone( );          // devuelve el tipo de objeto
```

Porque una matriz bidimensional es en realidad una matriz unidimensional que almacena otras matrices unidimensionales, la misma distinción entre una copia superficial y profunda Existe. Desafortunadamente, la clase java.util.Arrays no proporciona ningún "deepClone". método. Sin embargo, podemos implementar nuestro propio método clonando el individuo filas de una matriz, como se muestra en el Fragmento de Código 3.20, para una matriz bidimensional de números enteros.

```
1 público estático int[ ][ ] deepClone(int[ ][ ] original) {
2     int[ ][ ] backup = nuevo int[longitud original][ ]; para     // crea una matriz de matrices de nivel superior
3     (int k=0; k < longitud original; k++)
4         copia de seguridad[k] = original[k].clone();             // copiar la fila k
5 devolver copia de seguridad;
6 }
```

Fragmento de código 3.20: Un método para crear una copia profunda de una matriz bidimensional de números enteros.

3.6.2 Clonación de listas enlazadas

En esta sección, agregamos soporte para clonar instancias de la clase SinglyLinkedList De la Sección 3.2.1. El primer paso para que una clase sea clonable en Java es declarar que implementa la interfaz Cloneable. Por lo tanto, ajustamos la primera línea del La definición de clase aparecerá de la siguiente manera:

```
clase pública SinglyLinkedList<E> implementa Cloneable {
```

La tarea restante es implementar una versión pública del método clone() de la clase, que presentamos en el fragmento de código 3.21. Por convención, ese método debe comenzar creando una nueva instancia usando una llamada a super.clone(), que en nuestro El caso invoca el método de la clase Object (línea 3). Dado que la versión heredada devuelve un Object, realizamos una conversión de tipo SinglyLinkedList<E>.

En este punto de la ejecución, la otra lista se ha creado como una copia superficial. del original. Dado que nuestra clase de lista tiene dos campos, tamaño y encabezado, lo siguiente Se han realizado las siguientes asignaciones:

```
otro.tamaño = este.tamaño;
otro.cabeza = este.cabeza;
```

Si bien la asignación de la variable de tamaño es correcta, no podemos permitir que la nueva lista Comparten el mismo valor de cabecera (a menos que sea nulo). Para que una lista no vacía tenga un estado independiente, debe tener una cadena de nodos completamente nueva, cada una almacenando una referencia. al elemento correspondiente de la lista original. Por lo tanto, creamos una nueva cabecera. nodo en la línea 5 del código y luego realizar un recorrido por el resto del lista original (líneas 8 a 13) mientras se crean y vinculan nuevos nodos para la nueva lista.

```
1  público SinglyLinkedList<E> clone() lanza CloneNotSupportedException {
2    // utilice siempre Object.clone() heredado para crear la copia inicial
3    SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone(); // conversión segura
4    si (tamaño > 0) { // necesitamos una cadena de nodos independiente
5      otro.cabeza = nuevo Nodo<>(cabeza.getElement( ), null);
6      Node<E> walk = head.getNext( ); // recorre el resto de la lista original
7      Node<E> otherTail = other.head; // recordar el nodo creado más recientemente
8      mientras (walk != null) { // crea un nuevo nodo que almacena el mismo elemento
9        Nodo<E> más nuevo = nuevo Nodo<>(walk.getElement( ), null);
10       otherTail.setNext(más nuevo);                    // vincula el nodo anterior a este
11       otherTail = más nuevo;
12       caminar = caminar.getNext( );
13     }
14   }
15   devolver otro;
16 }
```

Fragmento de código 3.21: Implementación del método SinglyLinkedList.clone.