3.6 Cloning Data Structures

The beauty of object-oriented programming is that abstraction allows for a data structure to be treated as a single object, even though the encapsulated implementation of the structure might rely on a more complex combination of many objects. In this section, we consider what it means to make a copy of such a structure.

In a programming environment, a common expectation is that a copy of an object has its own state and that, once made, the copy is independent of the original (for example, so that changes to one do not directly affect the other). However, when objects have fields that are reference variables pointing to auxiliary objects, it is not always obvious whether a copy should have a corresponding field that refers to the same auxiliary object, or to a new copy of that auxiliary object.

For example, if a hypothetical AddressBook class has instances that represent an electronic address book—with contact information (such as phone numbers and email addresses) for a person's friends and acquaintances—how might we envision a copy of an address book? Should an entry added to one book appear in the other? If we change a person's phone number in one book, would we expect that change to be synchronized in the other?

There is no one-size-fits-all answer to questions like this. Instead, each class in Java is responsible for defining whether its instances can be copied, and if so, precisely how the copy is constructed. The universal Object superclass defines a method named **clone**, which can be used to produce what is known as a ***shallow copy*** of an object. This uses the standard assignment semantics to assign the value of each field of the new object equal to the corresponding field of the existing object that is being copied. The reason this is known as a shallow copy is because if the field is a reference type, then an initialization of the form duplicate.field = original.field causes the field of the new object to refer to the same underlying instance as the field of the original object.

A shallow copy is not always appropriate for all classes, and therefore, Java intentionally disables use of the clone( ) method by declaring it as **protected**, and by having it throw a CloneNotSupportedException when called. The author of a class must explicitly declare support for cloning by formally declaring that the class implements the Cloneable interface, and by declaring a public version of the clone( ) method. That public method can simply call the protected one to do the field-by-field assignment that results in a shallow copy, if appropriate. However, for many classes, the class may choose to implement a deeper version of cloning, in which some of the referenced objects are themselves cloned.

For most of the data structures in this book, we omit the implementation of a valid clone method (leaving it as an exercise). However, in this section, we consider approaches for cloning both arrays and linked lists, including a concrete implementation of the clone method for the SinglyLinkedList class.

3.6.1 Cloning Arrays

Although arrays support some special syntaxes such as a[k] and a.length, it is important to remember that they are objects, and that array variables are reference variables. This has important consequences. As a first example, consider the following code:

```
int[ ] data = {2, 3, 5, 7, 11, 13, 17, 19};
int[ ] backup;
backup = data;                          // warning; not a copy
```

The assignment of variable backup to data does not create any new array; it simply creates a new alias for the same array, as portrayed in Figure 3.23.

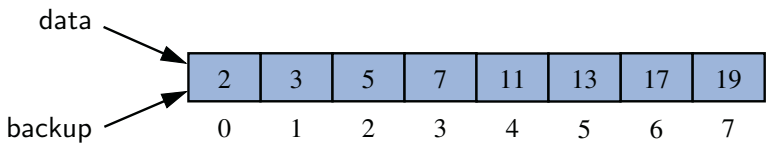

Figure 3.23: The result of the command backup = data for **int** arrays.

Instead, if we want to make a copy of the array, data, and assign a reference to the new array to variable, backup, we should write:

```
backup = data.clone( );
```

The clone method, when executed on an array, initializes each cell of the new array to the value that is stored in the corresponding cell of the original array. This results in an independent array, as shown in Figure 3.24.

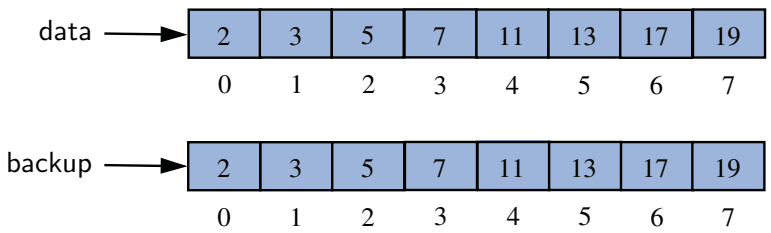

Figure 3.24: The result of the command backup = data.clone( ) for **int** arrays.

If we subsequently make an assignment such as data[4] = 23 in this configuration, the backup array is unaffected.

There are more considerations when copying an array that stores reference types rather than primitive types. The clone( ) method produces a ***shallow copy*** of the array, producing a new array whose cells refer to the same objects referenced by the first array.

For example, if the variable contacts refers to an array of hypothetical Person instances, the result of the command guests = contacts.clone( ) produces a shallow copy, as portrayed in Figure 3.25.

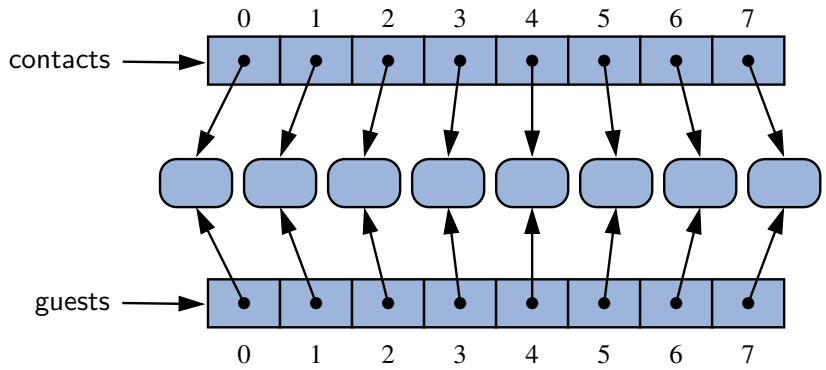

Figure 3.25: A shallow copy of an array of objects, resulting from the command guests = contacts.clone( ).

A ***deep copy*** of the contact list can be created by iteratively cloning the individual elements, as follows, but only if the Person class is declared as Cloneable.

```
Person[ ] guests = new Person[contacts.length];
for (int k=0; k < contacts.length; k++)
  guests[k] = (Person) contacts[k].clone( );        // returns Object type
```

Because a two-dimensional array is really a one-dimensional array storing other one-dimensional arrays, the same distinction between a shallow and deep copy exists. Unfortunately, the java.util.Arrays class does not provide any "deepClone" method. However, we can implement our own method by cloning the individual rows of an array, as shown in Code Fragment 3.20, for a two-dimensional array of integers.

```
1  public static int[ ][ ] deepClone(int[ ][ ] original) {
2    int[ ][ ] backup = new int[original.length][ ];      // create top-level array of arrays
3    for (int k=0; k < original.length; k++)
4      backup[k] = original[k].clone( );                  // copy row k
5    return backup;
6  }
```

Code Fragment 3.20: A method for creating a deep copy of a two-dimensional array of integers.

3.6.2 Cloning Linked Lists

In this section, we add support for cloning instances of the SinglyLinkedList class from Section 3.2.1. The first step to making a class cloneable in Java is declaring that it implements the Cloneable interface. Therefore, we adjust the first line of the class definition to appear as follows:

```
public class SinglyLinkedList<E> implements Cloneable {
```

The remaining task is implementing a public version of the clone( ) method of the class, which we present in Code Fragment 3.21. By convention, that method should begin by creating a new instance using a call to **super**.clone( ), which in our case invokes the method from the Object class (line 3). Because the inherited version returns an Object, we perform a narrowing cast to type SinglyLinkedList<E>.

At this point in the execution, the other list has been created as a shallow copy of the original. Since our list class has two fields, size and head, the following assignments have been made:

```
other.size = this.size;
other.head = this.head;
```

While the assignment of the size variable is correct, we cannot allow the new list to share the same head value (unless it is **null**). For a nonempty list to have an independent state, it must have an entirely new chain of nodes, each storing a reference to the corresponding element from the original list. We therefore create a new head node at line 5 of the code, and then perform a walk through the remainder of the original list (lines 8–13) while creating and linking new nodes for the new list.

```
1  public SinglyLinkedList<E> clone( ) throws CloneNotSupportedException {
2    // always use inherited Object.clone() to create the initial copy
3    SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone( ); // safe cast
4    if (size > 0) {                    // we need independent chain of nodes
5      other.head = new Node<>(head.getElement( ), null);
6      Node<E> walk = head.getNext( );  // walk through remainder of original list
7      Node<E> otherTail = other.head;  // remember most recently created node
8      while (walk != null) {           // make a new node storing same element
9        Node<E> newest = new Node<>(walk.getElement( ), null);
10       otherTail.setNext(newest);     // link previous node to this one
11       otherTail = newest;
12       walk = walk.getNext( );
13     }
14   }
15   return other;
16 }
```

Code Fragment 3.21: Implementation of the SinglyLinkedList.clone method.