

Cloning

APPENDIX 10

A **clone** of an object is an exact copy of the object. The emphasis here is on both *exact* and *copy*. A clone should have exactly the same data values as the original object, and it should be a distinct object and not simply another name for the original one.

A clone is made by invoking the method named `clone`. Recall from Chapter 8 that this method is defined in the class `Object` but needs to be overridden—that is, redefined before it will behave correctly for a specific class.

The standard class `ArrayList` is a class that defines its own `clone` method, as Chapter 12 mentioned. Suppose we define a list of strings using `ArrayList`, as follows:

```
ArrayList<String> aList = new ArrayList<String>();  
<Some code to fill aList>
```

We then can make an identical copy of `aList`, so that we have two separate copies, by invoking `ArrayList`'s method `clone`:

```
ArrayList<String> duplicateList =  
    (ArrayList<String>)aList.clone();
```

Although the method `clone` returns a copy of `aList`, it always returns it as an object of type `Object`. So we normally need a type cast.

For lists of objects other than strings, using the method `clone` can be more complicated and can lead to a few pitfalls. Let's examine this problem by first seeing how we can add a `clone` method to our own classes. For example, consider the class `Pet` in Listing 6.1 of Chapter 6. After we make suitable revisions to the class definition, we will be able to make a copy of an object of type `Pet` as follows:

```
Pet originalData = new Pet("Fido", 2, 5.6);  
Pet duplicateData = (Pet)originalData.clone();
```

Be sure to notice the type cast of the clone from `Object` to `Pet`. Now let's make the necessary revisions to `Pet`.

First, the class must implement the standard interface `Cloneable`. We do that by beginning the class definition for `Pet` as follows:

```
public class Pet implements Cloneable
```

This interface is actually empty, but it requires that you add a definition of the method `clone` to the class definition. The heading for the method `clone` in the class `Object` is

```
protected Object clone()
```

`Pet` will override this method with the following public version:

```
public Object clone()
```

If a class's instance variables have either primitive types or class types whose objects cannot be changed by their methods, such as the type `String`, the definition of `clone` need only invoke the inherited version of `clone`. Since `Pet` is such a class—it has one instance variable of type `String` and two that have primitive types—the method `clone` shown in Listing A10.1¹ will work fine. In that definition, the method `clone` invokes the version of `clone` in the class `Object`,² which simply makes a bit-by-bit copy of the memory used to store the calling object's instance variables. The try-catch blocks are required because the method `clone` can throw the exception `CloneNotSupportedException` if the class does not implement the `Cloneable` interface. Of course, in these classes, we are implementing the `Cloneable` interface, so the exception will never be thrown, but the compiler will still insist on the try-catch blocks.

LISTING A10.1 A Simple Implementation of the Method `clone`

```
public Object clone()
{
    try
    {
        return super.clone(); //Invocation of Object's clone
    }
    catch(CloneNotSupportedException e)
    { //This should not happen.
        return null; //To keep the compiler happy.
    }
}
```

Works correctly for a class like `Pet`, in which each instance variable has either a primitive type or the type `String`. Will not work correctly in most other cases.

1. The source code on the book's Website contains a version of the class `Pet` that includes this definition of the method `clone`.
2. If your class is a derived class of some class (other than `Object`), we are assuming that the base class has a well-defined `clone` method, since `super.clone` will then refer to the base class.

Note that the `Cloneable` interface and `clone` method behave in more complicated ways than most interfaces and inherited methods. Although the class `Object` has a method named `clone`, it is not inherited automatically. You must include `implements Cloneable` in the definition of the class, and you must include a definition of `clone`, even if it is simply defined as in Listing A10.1. This version of `clone` behaves as `clone` would if it were inherited from `Object` in the normal way.

If your class has instance variables of a class type whose objects can be changed by their methods, the definition of `clone` in Listing A10.1—although legal—probably will not do what you want a `clone` method to do. This version produces a `clone` that has a copy of each instance variable’s memory address, rather than a copy of the instance variable’s data. For a class like `String`, whose objects cannot be changed—that is, has no set methods—this condition is not a problem. Such was the case when we cloned a `Pet` object or the list of strings earlier in this appendix.

For most other classes, this condition would allow access to private data in the way we described in Section 6.5, entitled “Information Hiding Revisited,” of Chapter 6. For these classes, your definition of `clone` should make a clone of each instance variable of a changeable class type. Of course, this task requires that those class types for the instance variables have a suitable `clone` method themselves. The way to define such a `clone` method is illustrated in Listing A10.2. Let’s go over some of the details in that listing.

LISTING A10.2 A Class and Its `clone` Method

```
public class Neighbor implements Cloneable
{
    private String name;
    private int numberOfChildren;
    private Pet familyPet;

    public Object clone()
    {
        try
        {
            Neighbor copy = (Neighbor)super.clone();
            copy.familyPet = (Pet)familyPet.clone();
            return copy;
        }
        catch(CloneNotSupportedException e)
        {
            //This should not happen.
            return null; //To keep the compiler happy.
        }
    }
}
```

```

public Pet getPet()
{
    return (Pet)familyPet.clone();
}

<There are presumably other methods that are not shown.>
}

```

The class `Neighbor` has three instance variables. Two—`name` and `numberOfChildren`—are not a concern when we define a `clone` method, since their data types are `String` and `int`, respectively. But `familyPet`'s data type is `Pet`, which is a class that has set methods. Let's see how this affects `Neighbor`'s `clone` method.

The following statement in `clone` makes a bit-by-bit copy of the memory used to store the receiving object's instance variables:

```
Neighbor copy = (Neighbor)super.clone();
```

This sort of copy works fine for the instance variables `name` of type `String` and `numberOfChildren` of type `int`. However, the value it gives to `copy`'s instance variable `familyPet`—that is, `copy.familyPet` is the *address* of the instance variable `familyPet` in the receiving object. We want a *copy* of that instance variable, not its address. To get a copy, we call `Pet`'s `clone` method as follows:

```
copy.familyPet = (Pet)familyPet.clone();
```

`Neighbor`'s `clone` method then returns `copy` as the clone.

We have included `Neighbor`'s accessor method `getPet` in Listing A10.2. To avoid the privacy leak described in Section 6.5 of Chapter 6, `getPet` returns a clone of the instance variable `familyPet` instead of `familyPet` itself.

Let's return briefly to our opening example, where we cloned an object of `ArrayList`. This class's `clone` method does not clone the objects in the list. So while `aList` and `duplicateList` are two distinct objects, they share the same strings as their elements. For example, the first string in `aList` is also the first string in `duplicateList`. We do not have two separate but equal first strings. The good news is that the strings on the lists cannot be modified, so it is perfectly safe for the two lists to share the same collection of strings.