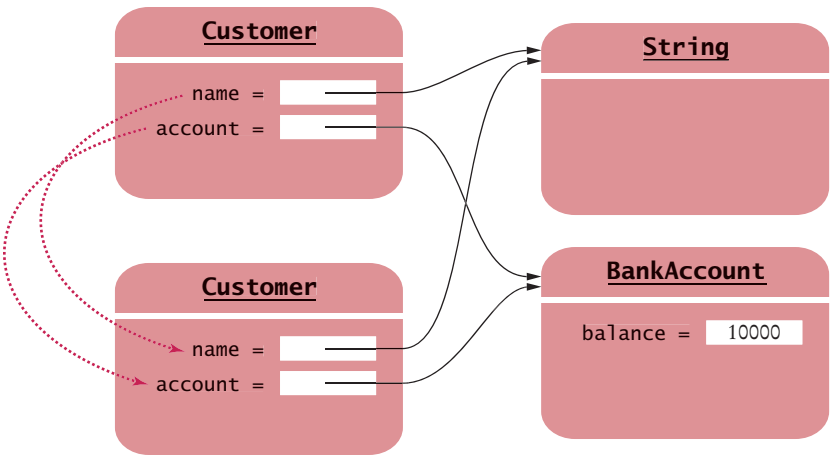
Special Topic 10.6

Implementing the `clone` Method

The `Object.clone` method is the starting point for the `clone` methods in your own classes. It creates a new object of the same type as the original object. It also automatically copies the instance variables from the original object to the cloned object. Here is a first attempt to implement the `clone` method for the `BankAccount` class:

```
public class BankAccount
{
   . . .
   public Object clone()
   {
      // Not complete
      Object clonedAccount = super.clone();
      return clonedAccount;
   }
}
```

However, this `Object.clone` method must be used with care. It only shifts the problem of cloning by one level; it does not completely solve it. Specifically, if an object contains a reference to another object, then the `Object.clone` method makes a copy of that object reference, not a clone of that object. The figure below shows how the `Object.clone` method works with a `Customer` object that has references to a `String` object and a `BankAccount` object. As you can see, the `Object.clone` method copies the references to the cloned `Customer` object and does not clone the objects to which they refer. Such a copy is called a **shallow copy**.



The `Object.clone` Method Makes a Shallow Copy

There is a reason why the `Object.clone` method does not systematically clone all subobjects. In some situations, it is unnecessary. For example, if an object contains a reference to a string, there is no harm in copying the string reference, because Java string objects can never change their contents. The `Object.clone` method does the right thing if an object contains only numbers, Boolean values, and strings. But it must be used with caution when an object contains references to other objects.

For that reason, there are two safeguards built into the `Object.clone` method to ensure that it is not used accidentally. First, the method is declared `protected` (see Special Topic 10.3 on page 439). This prevents you from accidentally calling `x.clone()` if the class to which `x` belongs hasn't declared `clone` to be public.

As a second precaution, `Object.clone` checks that the object being cloned implements the `Cloneable` interface. If not, it throws an exception. The `Object.clone` method looks like this:

```
public class Object
{
   protected Object clone()
         throws CloneNotSupportedException
   {
      if (this instanceof Cloneable)
      {
         // Copy the instance variables
         . . .
      }
      else
         throw new CloneNotSupportedException();
   }
}
```

Unfortunately, all that safeguarding means that the legitimate callers of `Object.clone()` pay a price—they must catch that exception (see Chapter 11) *even if their class implements* `Cloneable`.

```
public class BankAccount implements Cloneable
{
   . . .
   public Object clone()
   {
      try
      {
         return super.clone();
      }
      catch (CloneNotSupportedException e)
      {
         // Can't happen because we implement Cloneable but we still must catch it.
         return null;
      }
   }
}
```

If an object contains a reference to another mutable object, then you must call `clone` for that reference. For example, suppose the `Customer` class has an instance variable of class `BankAccount`. You can implement `Customer.clone` as follows:

```
public class Customer implements Cloneable
{
   private String name;
   private BankAccount account;
   . . .
   public Object clone()
   {
      try
      {
```

```
            Customer cloned = (Customer) super.clone();
            cloned.account = (BankAccount) account.clone();
            return cloned;
         }
         catch(CloneNotSupportedException e)
         {
            // Can't happen because we implement Cloneable
            return null;
         }
      }
   }
```