How to avoid traps and correctly override methods from java.lang.Object

Avoid incorrect implementations and bugs by following these guidelines

Summary
The base class for all Java classes, java.lang.Object allows five of its methods to be overridden by subclasses. Sometimes it is necessary to override the default implementations provided by Object. Unfortunately, it's easy to override these methods incorrectly, which can result in classes that are difficult to subclass and that fail, in subtle ways, to work properly. This article explains why it's important to implement these methods correctly and also shows how to override these methods and get it right. (3,500 words)

By Mark Roulo

All Java classes eventually have java.lang.Object, hereafter referred to simply as Object, as a base class. Because of this, all Java classes inherit methods from Object. Half of these methods are final and cannot be overridden. However, the other methods in Object can be and are overridden, often incorrectly. This article explains why it's important to implement these methods correctly and then explains how to do so.

Object declares three versions of the wait method, as well as the methods notify, notifyAll and getClass. These methods all are final and cannot be overridden. This article discusses the remaining methods that are not final and that often must be overridden: clone, toString, equals, hashCode, finalize.

I'll discuss the clone method first, as it provides a nice collection of subtle traps without being excessively complicated. Next, I'll consider equals and hashCode together. These are the most difficult to implement correctly. Wrapping up the article, I'll describe how to override the comparatively simple toString and finalize methods.

Why this matters
Why is it important to implement these methods correctly? In a small application written, used, and maintained by one individual, it may not be important. However, in large applications, in applications maintained by many people and in libraries intended for use by other people, failing to implement these methods correctly can result in classes that cannot be subclassed easily and that do not work as expected.

It is, for example, possible to write the clone method so that no child classes can be cloned. This will be a problem for users who want to extend the class with the improperly written clone method. For in-house development this mistake can result in excess debug time and rework when the problem is finally discovered. If the class is provided as part of a class library you sell to other programmers, you may find yourself rereleasing your library, handling excess technical support calls, and possibly losing sales as customers discover that your classes can't be extended.

Erroneous implementations of equals and hashCode can result in losing elements stored in hashtables. Incorrect implementation of these methods can also result in intermittent, data-dependent bugs as behavior changes over time. Again, this can result in excess debugging and extra software releases, technical support calls, and possibly lost sales. Implementing toString improperly is the least damaging, but can still result in loss of time, as you must debug if the name of the object is wrong.

In short, implementing these methods incorrectly can make it difficult or impossible for other programmers to subclass and use the classes with the erroneous implementation. Less serious, but still important, implementing these methods incorrectly can result in time lost to debugging.

Two themes
Two themes will reappear throughout this article. The first theme is that you must pay attention to whether your implementations of these methods will continue to be correct in child classes. If not, you should either rewrite your implementations to be correct in child classes or declare your class to be final so that there are no child classes.

The second theme is that methods have contracts -- defined behavior -- and when implementing or overriding a method, the contract should be fulfilled. The equals method of Object provides an example of a contract: the contract states that if the parameter to equals is null, then equals must return false. When overriding equals, you are responsible for ensuring that all the specifics of the contract are still met.

Implementing clone
The clone method allows clients to obtain a copy of a given object without knowing the precise class of the original object. The clone method in Object is a magic function that generates a shallow copy of the entire object being cloned.

To enable shallow cloning of your class, you implement the Cloneable interface. (For a full discussion of shallow copying versus deep copying, see the sidebar below.) Since Cloneable is a tagging interface with no methods, it's simple to implement:

```java
public class BaseClass implements Cloneable {
    // Rest of the class.
}
```
Notice that you don't even have to write the clone method!

}  

clone is a protected method. If you want objects from other packages to be able to call it, you must make clone public. You do this by redéclaring clone and then calling the superclass's clone method:

public class BaseClass implements Cloneable
{
    // Rest of the class.

    public Object clone ()
    throws CloneNotSupportedException
    {
        return super.clone();
    }
}

Finally, if you want some of the member data in the class to be copied deeply, you must copy these members yourself:

public class BaseClass implements Cloneable
{
    // SomeOtherClass is just an example. It might look like
    // this:
    //
    //     class SomeOtherClass implements Cloneable
    //     {
    //         public Object clone ()
    //             throws CloneNotSupportedException
    //             {
    //                 return super.clone();
    //             }
    //     }
    //
    private SomeOtherClass data;

    // Rest of the class.

    public Object clone ()
    throws CloneNotSupportedException
    {
        BaseClass newObject = (BaseClass)super.clone();

        // At this point, newObject shares the SomeOtherClass
        // object referred to by this.data with the object
        // running clone. If you want newObject to have its own
        // copy of data, you must clone this data yourself.

        if (this.data != null)
            newObject.data = (SomeOtherClass)this.data.clone();

        return newObject;
    }
}

That's it. So, what mistakes should you look out for?

Don't fail to implement the Cloneable interface if you want your class to be cloneable. The clone method from Object checks that the Cloneable interface has been implemented. If the Cloneable interface hasn't been implemented, a CloneNotSupportedException is thrown when clone is called.

Don't implement clone by using a constructor. The javadoc for the clone method states that it:
Creates a new object of the same class as this object. It then initializes each of the new object's fields by assigning it the same value as the corresponding field in this object. No constructor is called.

Notice that "no constructor is called." Avoid implementing clone as follows:

```java
public class BaseClass implements Cloneable {
    public BaseClass (/* parameters */) {
        // Code goes here...
    }
    // Rest of the class.
    public Object clone ()
        throws CloneNotSupportedException {
        return new BaseClass (/* parameters */);
    }
}
```

There are two reasons to avoid such an approach: First, the contract for clone states that no constructor is called. Second, and more importantly, child classes now return the wrong type from clone. In the example below, the object returned by clone is a BaseClass, not a ChildClass!

```java
public class ChildClass extends BaseClass {
    // Use clone from BaseClass
}
```

Further, the child class cannot override clone to make a deep copy of the member variables in the ChildClass. The following code demonstrates this problem:

```java
public class ChildClass extends BaseClass {
    private SomeOtherClass data;
    // Rest of the class.
    public Object clone () throws CloneNotSupportedException {
        // The cast in the line below throws an exception!
        // ChildClass newObject = (ChildClass)super.clone();

        // You _never_ get here because the line above throws
        // an exception.
        if (this.data != null)
            newObject.data = (SomeOtherClass)this.data.clone();

        return newObject;
    }
}
```

The first line in clone throws an exception because the clone method in BaseClass returns a BaseClass object not a ChildClass object.

Summary: Don't implement clone by using a copy constructor.

Avoid using constructors to copy subobjects when possible. Another mistake is to use constructors to copy subobjects when implementing clone. Consider the following example class, which uses Dimension as the subobject:

```java
import java.awt.Dimension;
```
public class Example implements Cloneable
{
    private Dimension dim;

    public void setDimension (Dimension dim)
    {
        this.dim = dim;
    }

    public Object clone () throws CloneNotSupportedException
    {
        Example newObject = (Example)super.clone();

        // Notice the use of a constructor below instead of
        // a clone method call. If you have a sub-class of
        // Dimension, any data in the sub-class (e.g. a third
        // dimension value like z) will be lost.
        //
        if (this.dim != null)
            newObject.dim = new Dimension (dim);

        return newObject;
    }
}

If a child class of Dimension is passed to setDimension, the object returned by clone will be different from the original object. The preferred way to write this clone method would be:

import java.awt.Dimension;

public class Example implements Cloneable
{
    private Dimension dim;

    public void setDimension (Dimension dim)
    {
        this.dim = dim;
    }

    public Object clone () throws CloneNotSupportedException
    {
        Example newObject = (Example)super.clone();

        // Call 'this.dim.clone()' instead of
        // 'new Dimension(dim)'
        //
        if (this.dim != null)
            newObject.dim = (Dimension)this.dim.clone();

        return newObject;
    }
}

Now, if a child class of Dimension is passed to setDimension, it is copied properly when clone is called. Unfortunately, while the preferred code above compiles under the Java 2 platform (formerly known as JDK 1.2), it won't compile under JDK 1.1.7. Dimension doesn't implement Cloneable in JDK 1.1 and the clone method for Dimension is protected so Example can't call it anyway. This means that under JDK 1.1 you must write Example's clone method using a copy constructor.
for the Dimension member variable even though you don't want to. If a child of Dimension is passed to setDimension, you'll have a problem if you try to clone an Example object.

Testing explicitly for Dimension in the clone method is one workaround:

```java
import java.awt.Dimension;

public class Example implements Cloneable {

  private Dimension dim;

  public void setDimension (Dimension dim) {
    this.dim = dim;
  }

  public Object clone () throws CloneNotSupportedException {
    Example newObject = (Example)super.clone();
    if (this.dim != null) {
      // Test explicitly for Dimension here.  Don't test
      // using the instanceof operator -- it doesn't do
      // what you want it to.
      if (this.dim.getClass() != Dimension.class)
        throw new CloneNotSupportedException("Wrong sub-class for 'dim'");

      newObject.dim = new Dimension (dim);
    }
    return newObject;
  }
}
```

This is better than returning a clone object with the wrong data for dim, but it still isn't a good solution.

Summary: Make copies of member variables using their clone methods if possible.

Pay attention to synchronization on the clone method. clone is a method just like any other. In a multithreaded environment you want to synchronize clone so that the underlying object stays internally consistent while being copied. You must then also synchronize the mutator methods. Note that this is different from a constructor, which almost never needs synchronization. Sometimes you should treat clone like a constructor. Even though the clone method isn't a constructor, sometimes you should treat it like one. If you do something special in each constructor, like incrementing an "objects created" count, you probably want to do the same thing in the clone method.

Classes used by others should usually implement clone. This is most important when the class is part of a class library used by others who don't have access to the source code. Failing to implement the clone method can cause problems for clients attempting to write their own clone methods -- see the problems with Dimension in (2) above. If you're producing a third-party library, don't force your customers to work around a lack of cloning.

If you're not producing a third-party library, waiting to implement clone until it's needed for each class is reasonable. This is especially true because once you've overridden clone, you must pay careful attention to overriding clone in all the child classes. Child classes must pay attention to clone methods inherited from parent classes. Well-written third-party library classes will often implement clone. However, once a class becomes cloneable, that class's children become cloneable, too. If you extend a class that is cloneable, you must consider whether the clone method you inherit (which will make a shallow copy of all the data in your subclass) does what you want it to. If it doesn't, you must override clone.

**Implementing equals and hashCode**

Because of their contracts, if you override either the equals or hashCode methods from Object, you must almost certainly override the other method as well. The complicated contracts that go with these methods make overriding them correctly very difficult. Some of the code shipped with the standard Java libraries gets it wrong.

Here are the important contract requirements for the two methods, as documented in the javadoc documentation for java.lang.Object:
The `hashCode` method must return the same integer value every time it is invoked on the same object during the entire execution of a Java application or applet. It need not return the same value for different runs of an application or applet. The Java 2 platform (Java 2) documentation further allows the `hashCode` value to change if the information used in the `equals` method changes.

If two objects are equal according to the `equals` method, they must return the same value from `hashCode`. The `hashCode` value to change if the information used in the `equals` method changes.

The `hashCode` value to change if the information used in the `equals` method changes.

The `hashCode` value to change if the information used in the `equals` method changes.

Finally, `x.equals(null)` should return false.

Object provides a simple implementation of `equals`. It just tests the two objects for referential equality: does `x` equal `y`? Some of the standard Java classes override this to provide a more useful notion of equality -- usually content equality (i.e., is some or all of the data in the two objects identical?).

The `hashCode` implementation of `java.lang.String`, for example, returns true if the two objects are both `String` objects containing exactly the same characters in exactly the same order. The `hashCode` method of `java.awt.Dimension` returns true if the passed-in object is a dimension with the same width and height as the `Dimension` object executing the `equals` method.

The default implementation of `hashCode` provided by `Object` returns something corresponding to the object's address in memory or location in the Java virtual machine's global object array. Again, some of the standard Java classes override this method. `String`, for example, overrides the `hashCode` implementation in `Object` to return a hash of some or all of the characters making up the `String`. This allows two `String` objects with the same characters in the same order to return the same hash value. `Dimension` uses the `hashCode` method provided by `Object`.

Now for the bad news: It's almost impossible to override `equals` and `hashCode` for mutable classes and provide useful, correct and safe implementations for both methods.

To see why, consider the class `java.awt.Dimension`. This class overrides `equals`, but not `hashCode`. `Dimension`'s JDK 1.1 implementation of `equals` looks like this:

```java
public boolean equals(Object obj) {
    if (obj instanceof Dimension) {
        Dimension d = (Dimension)obj;
        return (width == d.width) && (height == d.height);
    }
    return false;
}
```

This is a fairly reasonable implementation of content equality: if two `Dimension` objects have the same width and height they're equal, otherwise they aren't. So, what's wrong?

The first problem is that because `Dimension` doesn't override `hashCode`, it's possible to have two `Dimension` objects that are equal, but return different `hashCode` values. This violates requirement (2) from above.

Second, testing the input parameter using `instanceof Dimension` creates problems of its own. Consider a child class: `ThreeDeeDimension`. Objects of type `ThreeDeeDimension` should test as equal only if they have identical height, width and depth. `ThreeDeeDimension` might look like this:

```java
import java.awt.Dimension;

public class ThreeDeeDimension extends Dimension {
    // I don't like public data, but I'll make this public
    // to be similar to Dimension's width and height.
    //
    public int depth;

    public ThreeDeeDimension (int width, int height, int depth) {
        super (width, height);
        this.depth = depth;
    }
}
```
public boolean equals (Object o)
{
    if ((super.equals (o) == true) && (o.getClass().equals(this.getClass())))
        return ((ThreeDeeDimension)o).width == this.width;
    else
        return false;
}
}

Unfortunately, this implementation of equals doesn't meet requirement (4) listed above. The following code snippet shows this:

import java.awt.Dimension;

public class Main
{
    static public void main (String[] args)
    {
        Dimension         dim         = new Dimension (1, 2);
        ThreeDeeDimension threeDeeDim = new ThreeDeeDimension (1, 2, 3);

        // This will print out that the two objects are equal.
        System.out.println ("dim.equals(threeDeeDim) = " + dim.equals(threeDeeDim));

        // And this will print out that the two objects are not equal.
        System.out.println ("threeDeeDim.equals(dim) = " + threeDeeDim.equals(dim));

        // And requirement (4) is that both tests should return the
        // same result.
    }
}

I can fix this problem by rewriting the equals method of Dimension. If I write equals in Dimension like this, the ThreeDeeDimension class above meets requirement (4):

    public boolean equals (Object obj)
    {
        if (obj != null && (obj.getClass().equals(this.getClass())))
            
            Dimension d = (Dimension)obj;
            return (width == d.width) && (height == d.height);
        }
        return false;
    }

Now, objects of type ThreeDeeDimension won't return true when compared to objects of type Dimension. You still have a problem with both Dimension and ThreeDeeDimension because they don't meet requirement (2): objects that test as equal should have identical hashCode values. So, how is content-equality implemented in mutable classes? One example with both equals and hashCode is:

    public class MutableExample
    {
        private int x;

        // Constructors and other methods.

        public boolean equals (Object o)
{  
// Test for null to meet requirement (7) and also to  
// avoid a NullPointerException when calling getClass()  
// below.  
//  
// Comparing classes ensures that parent class objects won't test  
// equal to child class objects. Parent objects should almost  
// never test equal to child objects. This makes meeting requirements  
// (4) and (5) possible.  
//  
// if ((o != null) && (o.getClass().equals(this.getClass())))  
//  
// If you were inheriting directly from a class that  
// overrode equals, you would insert a line here that  
// looked like this:  
// if (super.equals (o) == false)  
//    return false;  
//  
// If overriding the equals method from Object, don't  
// call super.equals().  
//  
// This is the point. We have already tested the equality  
// of our parent class. Now we test for equality of the  
// data added by _this_ child class. This also meets requirement (3).  
    
    return ((MutableExample)o).x == this.x;  
}  
return false;  
}  

public int hashCode ()  
{  
// This meets requirements (1) and (2). You always return the  
// same value for each object because you always return the  
// same value for all objects. You also return identical  
// hashCode values when two objects test as equals because  
// you always return identical hashCode values. There is no  
// requirement to return different hashCode values when  
// two objects test as not equal.  
//  
// The only real problem with this implementation is that it  
// is an almost totally useless implementation of hashCode.  
// It can turn a Hashtable lookup into a linear search.  
//  
// With JDK 1.1 you can't return 'x', because it can change  
// and the requirements for hashCode are that the same value  
// must be returned each time hashCode is called on the same object.  
//  
// Java 2 (formerly JDK 1.2) allows 'return x'; because Java 2 allows the hashCode  
// value to change if the underlying data changes. This is more  
// friendly, but still allows data to be lost in hashtables  
// if the underlying hashCode value changes.  
//  
return 0;
This implementation meets all the requirements, including requirement (6) with the clarification provided by Java 2. Immutable classes make implementing a useful and safe `hashCode` easier. In this case, you can use the data in the class to generate a hash value because that data will never change. In the example above, if “x” was guaranteed to never change, I could have implemented `hashCode` like this:

```java
public int hashCode()
{
    // Legal, useful and safe, but only because 'x' never changes.
    //
    return x;
}
```

The key points to remember when implementing `equals` and `hashCode`:

These are not simple methods to implement. There are many details specified in each method's contract. You must implement these two methods together. You can rarely implement just one of them. It is difficult to implement a correct, useful, and safe `hashCode` method for mutable classes. Making classes immutable makes implementing the `hashCode` and `equals` methods much easier. Java 2 allows the value returned by `hashCode` to change if the underlying data changes, but you should be wary of doing this because data can then be stranded in hashtables.

You must pay attention to inheritance, especially when implementing `equals`. This means comparing classes with `getClass` rather than with `instanceof`.

Once a class has overridden `equals` and `hashCode`, the child classes may also require their own implementations.

**Implementing `toString`**

The `toString` method is the easiest of all the methods in `Object` to override correctly. This is because the contract is so loosely defined. The javadoc for `toString` reads:

> [toString] returns a string representation of the object. In general, the `toString` method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

The `toString` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@`, and the unsigned hexadecimal representation of the hash code of the object.

All of the above requirements are fuzzy. The method must return a "string representation" that in general "textually represents" the objects and should be "concise," but "informative." None of these requirements are as precise as the requirements for `clone`, `hashCode` or `equals`. Nevertheless, it is still possible to implement this method somewhat incorrectly. Consider the following example:

```java
public class BaseClass
{
    private int x;

    // Constructors and other member data and methods ...

    public String toString()
    {
        // This implementation is not quite correct.

        return "BaseClass[" + x + "]";
    }
}
```

Calling `toString` on objects of this class will result in output that looks something like this (assuming `x` equals 4):

```
BaseClass[4]
```

The problem here is that someone might extend `BaseClass` and might not override `toString`. An example of this is:

```java
public class Extension extends BaseClass
{
    // Constructors, member data and methods ...
}
```
Now, calling `toString` on objects of class `Extension` results in output that looks like this:

```
BaseClass[4]
The class name reported by `toString` is wrong! The object is an `Extension` object and the `toString` method is reporting it as a `BaseClass` object. You could blame the `Extension` class for not implementing `toString` itself, but the contract for `toString` only recommends that child classes implement their own version. There is no requirement that child classes do so. Instead, you should write `toString` in `BaseClass` so that it behaves correctly in child classes. You can do this by writing the `toString` implementation like this:

```java
public String toString()
{
    // This implementation behaves properly in child classes.
    return getClass().getName() + "[" + x + "]";
}
```

Now calling `toString` on objects of class `Extension` results in this output:

```
Extension[4]
```

which is correct.

**Implementing finalize**
There are only three relatively simple things to remember if you choose to override `finalize`. First, you should call the `finalize` method of the parent class in case it has cleanup to do.

```java
protected void finalize() throws Throwable
{
    super.finalize();
}
```

Second, you should not depend on the `finalize` method being called. There is no guarantee of when (or if) objects will get garbage collected and thus no guarantee that the `finalize` method will be called before the running program exits. Finally, remember that code in `finalize` might fail and throw exceptions. You may want to catch these so the `finalize` method can continue.

```java
protected void finalize() throws Throwable
{
    try
    {
        // Do stuff here to clean up your object.
    }
    catch (Throwable t)
    {
    }
    super.finalize();
}
```

In general, `finalize` doesn't need to be overridden.

**Conclusion**
There are traps to overriding all of the nonfinal methods inherited from `java.lang.Object`. Some of them are subtle enough that even classes provided in the core Java libraries get them wrong. Nevertheless, with a bit of care, you can implement the methods correctly.

When building large products and when constructing third-party class libraries, it is important to take the care needed to get these implementations right. After all, some developer might read the documentation and assume your code does what the documentation says. Failing to implement these methods correctly for large projects can result in extra time spent debugging. When implementing these methods in libraries sold commercially, it is even more important to implement these methods correctly; you cannot easily fix things once the library has been released. Failing to implement these methods properly can result in your library being harder to use and extend than it should be. Finally, for smaller projects, it can sometimes be reasonable to meet most, but not all, of the requirements for these methods. In those cases you should at least make your decision consciously and document it.

**Shallow copying versus deep copying**
Shallow copying means that member variables are copied to the clone object while subobjects only have their references copied – so no new subobjects are created. An example is:

```java
import java.awt.Dimension;
public class Example implements Cloneable{
    private Dimension dim = new Dimension();
    public synchronized Object clone () throws CloneNotSupportedException{
        return super.clone();
    }
    public synchronized void setWidth (int width)
    {
        dim.width = width;
    }
    public synchronized int getWidth ()
    {
        return dim.width;
    }
}
```

Creating an object of type Example and then cloning it will result in only one object of type Dimension, to be shared by both Example objects. The example below shows this:

```java
public class Main
{
    public static void main (String[] args)
    {
        try
        {
            Example e1 = new Example();
            System.out.println("e1 width = " + e1.getWidth());
            Example e2 = (Example)e1.clone();
            System.out.println("e2 width = " + e2.getWidth());
            System.out.println("Setting e1 width to 10");
            e1.setWidth (10);
            System.out.println("e1 width = " + e1.getWidth());
            System.out.println("e2 width = " + e2.getWidth());
        }
        catch (CloneNotSupportedException exception)
        {
            System.out.println("Ooops");
        }
    }
}
```

When the example above is run, the output is:
```
e1 width = 0
e2 width = 0
Setting e1 width to 10
e1 width = 10
e2 width = 10
```

Notice that setting the width in e1 also sets the width in e2 because they're sharing the underlying Dimension object.

Deep copying means that each object gets its own copy of the subobjects. If you want clone to implement a deep copy, you must perform this copying yourself. To make the clone method for Example return a deep copy, you would write clone like this:

```java
import java.awt.Dimension;
public class Example implements Cloneable{
    private Dimension dim = new Dimension();
    public synchronized Object clone () throws
```
CloneNotSupportedException
{
    Example copy = (Example)super.clone();
    copy.dim = new Dimension(dim);
    return copy;
}

public synchronized void setWidth (int width)
{
    dim.width = width;
}
public synchronized int getWidth ()
{
    return dim.width;
}

When you run the example with the new version of clone, the output is:
  e1 width = 0
  e2 width = 0
  Setting e1 width to 10
  e1 width = 10
  e2 width = 0
**Use serialization to make deep copies and avoid extensive manual editing or extending of classes**

Implementing a deep copy of an object can be a learning experience -- you learn that you don't want to do it! If the object in question refers to other complex objects, which in turn refer to others, then this task can be daunting indeed. Traditionally, each class in the object must be individually inspected and edited to implement the Cloneable interface and override its clone() method in order to make a deep copy of itself as well as its contained objects. This article describes a simple technique to use in place of this time-consuming conventional deep copy.

**The concept of deep copy**

In order to understand what a deep copy is, let's first look at the concept of shallow copying. Mark Roulo explains how to clone objects as well as how to achieve shallow copying instead of deep copying. To summarize briefly here, a shallow copy occurs when an object is copied without its contained objects. To illustrate, Figure 1 shows an object, obj1, that contains two objects, containedObj1 and containedObj2.

![Figure 1. The original state of obj1](image1)

If a shallow copy is performed on obj1, then it is copied but its contained objects are not, as shown in Figure 2.

![Figure 2. After a shallow copy of obj1](image2)

A deep copy occurs when an object is copied along with the objects to which it refers. Figure 3 shows obj1 after a deep copy has been performed on it. Not only has obj1 been copied, but the objects contained within it have been copied as well.
Figure 3. After a deep copy of obj1

If either of these contained objects themselves contain objects, then, in a deep copy, those objects are copied as well, and so on until the entire graph is traversed and copied.

Each object is responsible for cloning itself via its clone() method. The default clone() method, inherited from Object, makes a shallow copy of the object. To achieve a deep copy, extra logic must be added that explicitly calls all contained objects' clone() methods, which in turn call their contained objects' clone() methods, and so on. Getting this correct can be difficult and time consuming, and is rarely fun. To make things even more complicated, if an object can't be modified directly and its clone() method produces a shallow copy, then the class must be extended, the clone() method overridden, and this new class used in place of the old. (For example, Vector does not contain the logic necessary for a deep copy.) And if you want to write code that defers until runtime the question of whether to make a deep or shallow copy an object, you're in for an even more complicated situation. In this case, there must be two copy functions for each object: one for a deep copy and one for a shallow. Finally, even if the object being deep copied contains multiple references to another object, the latter object should still only be copied once. This prevents the proliferation of objects, and heads off the special situation in which a circular reference produces an infinite loop of copies.

Serialization

Back in January of 1998, JavaWorld initiated its JavaBeans column by Mark Johnson with an article on serialization, "Do it the 'Nescafé' way -- with freeze-dried JavaBeans." To summarize, serialization is the ability to turn a graph of objects (including the degenerate case of a single object) into an array of bytes that can be turned back into an equivalent graph of objects. An object is said to be serializable if it or one of its ancestors implements java.io.Serializable or java.io.Externalizable. A serializable object can be serialized by passing it to the writeObject() method of an ObjectOutputStream object. This writes out the object's primitive data types, arrays, strings, and other object references. The writeObject() method is then called on the referred objects to serialize them as well. Further, each of these objects have their references and objects serialized; this process goes on and on until the entire graph is traversed and serialized. Does this sound familiar? This functionality can be used to achieve a deep copy.

Deep copy using serialization

1. The steps for making a deep copy using serialization are:
2. Ensure that all classes in the object's graph are serializable.
3. Create input and output streams.
4. Use the input and output streams to create object input and object output streams.
5. Pass the object that you want to copy to the object output stream.
6. Read the new object from the object input stream and cast it back to the class of the object you sent.

I have written a class called ObjectCloner that implements steps two through five. The line marked "A" sets up a ByteArrayOutputStream which is used to create the ObjectOutputStream on line B. Line C is where the magic is done. The writeObject() method recursively traverses the object's graph, generates a new object in byte form, and sends it to the ObjectOutputStream. Line D ensures the whole object has been sent. The code on line E then creates a ByteArrayInputStream and populates it with the contents of the ByteArrayOutputStream. Line F instantiates an ObjectInputStream using the ByteArrayInputStream created on line E and the object is deserialized and returned to the calling method on line G. Here's the code:
import java.io.*;
import java.util.*;
import java.awt.*;
public class ObjectCloner
{
    // so that nobody can accidentally create an ObjectCloner object
    private ObjectCloner()
    {
    }
    // returns a deep copy of an object
    static public Object deepCopy(Object oldObj) throws Exception
    {
        ObjectOutputStream oos = null;
        ObjectInputStream ois = null;
        try
        {
            ByteArrayOutputStream bos =
                new ByteArrayOutputStream(); // A
            oos = new ObjectOutputStream(bos); // B
            // serialize and pass the object
            oos.writeObject(oldObj);   // C
            oos.flush();               // D
            ByteArrayInputStream bin =
                new ByteArrayInputStream(bos.toByteArray()); // E
            ois = new ObjectInputStream(bin);                   // F
            // return the new object
            return ois.readObject(); // G
        }
        catch(Exception e)
        {
            System.out.println("Exception in ObjectCloner = " + e);
            throw(e);
        }
        finally
        {
            oos.close();
            ois.close();
        }
    }
}

All a developer with access to ObjectCloner is left to do before running this code is ensure that all classes in
the object's graph are serializable. In most cases, this should have been done already; if not, it ought to be
relatively easy to do with access to the source code. Most of the classes in the JDK are serializable; only the
ones that are platform-dependent, such as FileDescriptor, are not. Also, any classes you get from a third-party
vendor that are JavaBean-compliant are by definition serializable. Of course, if you extend a class that is
serializable, then the new class is also serializable. With all of these serializable classes floating around,
chances are that the only ones you may need to serialize are your own, and this is a piece of cake compared to
going through each class and overwriting clone() to do a deep copy.
An easy way to find out if you have any nonserializable classes in an object's graph is to assume that they are
all serializable and run ObjectCloner's deepCopy() method on it. If there is an object whose class is not
serializable, then a java.io.NotSerializableException will be thrown, telling you which class caused the problem.
A quick implementation example is shown below. It creates a simple object, v1, which is a Vector that
contains a Point. This object is then printed out to show its contents. The original object, v1, is then copied to
a new object, vNew, which is printed to show that it contains the same value as v1. Next, the contents of v1 are changed, and finally both v1 and vNew are printed so that their values can be compared.

```java
import java.util.*;
import java.awt.*;
public class Driver1
{
    static public void main(String[] args)
    {
        try
        {
            // get the method from the command line
            String meth;
            if((args.length == 1) &&
                ((args[0].equals("deep")) || (args[0].equals("shallow"))))
            {
                meth = args[0];
            }
            else
            {
                System.out.println("Usage: java Driver1 [deep, shallow]");
                return;
            }
            // create original object
            Vector v1 = new Vector();
            Point p1 = new Point(1,1);
            v1.addElement(p1);
            // see what it is
            System.out.println("Original = "+v1);
            Vector vNew = null;
            if(meth.equals("deep"))
            {
                // deep copy
                vNew = (Vector)(ObjectCloner.deepCopy(v1));   // A
            }
            else if(meth.equals("shallow"))
            {
                // shallow copy
                vNew = (Vector)v1.clone();                   // B
            }
            // verify it is the same
            System.out.println("New = "+vNew);
            // change the original object's contents
            p1.x = 2;
            p1.y = 2;
            // see what is in each one now
            System.out.println("Original = "+v1);
            System.out.println("New = "+vNew);
        }
        catch(Exception e)
        {
            System.out.println("Exception in main = "+e);
        }
    }
}
```
To invoke the deep copy (line A), execute java.exe Driver1 deep. When the deep copy runs, we get the following printout:

```
Original = [java.awt.Point[x=1,y=1]]
New      = [java.awt.Point[x=1,y=1]]
Original = [java.awt.Point[x=2,y=2]]
New      = [java.awt.Point[x=1,y=1]]
```

This shows that when the original Point, p1, was changed, the new Point created as a result of the deep copy remained unaffected, since the entire graph was copied. For comparison, invoke the shallow copy (line B) by executing java.exe Driver1 shallow. When the shallow copy runs, we get the following printout:

```
Original = [java.awt.Point[x=1,y=1]]
New      = [java.awt.Point[x=1,y=1]]
Original = [java.awt.Point[x=2,y=2]]
New      = [java.awt.Point[x=2,y=2]]
```

This shows that when the original Point was changed, the new Point was changed as well. This is due to the fact that the shallow copy makes copies only of the references, and not of the objects to which they refer. This is a very simple example, but I think it illustrates the, um, point.

**Implementation issues**

Now that I've preached about all of the virtues of deep copy using serialization, let's look at some things to watch out for.

The first problematic case is a class that is not serializable and that cannot be edited. This could happen, for example, if you're using a third-party class that doesn't come with the source code. In this case you can extend it, make the extended class implement `Serializable`, add any (or all) necessary constructors that just call the associated superconstructor, and use this new class everywhere you did the old one (here is an example of this).

This may seem like a lot of work, but, unless the original class's `clone()` method implements deep copy, you will be doing something similar in order to override its `clone()` method anyway.

The next issue is the runtime speed of this technique. As you can imagine, creating a socket, serializing an object, passing it through the socket, and then deserializing it is slow compared to calling methods in existing objects. Here is some source code that measures the time it takes to do both deep copy methods (via serialization and `clone()`) on some simple classes, and produces benchmarks for different numbers of iterations.

The results, shown in milliseconds, are in the table below:

<table>
<thead>
<tr>
<th>Procedure\Iterations(n)</th>
<th>1000</th>
<th>10000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>clone</td>
<td>10</td>
<td>101</td>
<td>791</td>
</tr>
<tr>
<td>serialization</td>
<td>1832</td>
<td>11346</td>
<td>107725</td>
</tr>
</tbody>
</table>

As you can see, there is a large difference in performance. If the code you are writing is performance-critical, then you may have to bite the bullet and hand-code a deep copy. If you have a complex graph and are given one day to implement a deep copy, and the code will be run as a batch job at one in the morning on Sundays, then this technique gives you another option to consider.

Another issue is dealing with the case of a class whose objects' instances within a virtual machine must be controlled. This is a special case of the Singleton pattern, in which a class has only one object within a VM. As discussed above, when you serialize an object, you create a totally new object that will not be unique. To get around this default behavior you can use the `readResolve()` method to force the stream to return an appropriate object rather than the one that was serialized. In this particular case, the appropriate object is the same one that was serialized. Here is an example of how to implement the `readResolve()` method. You can find out more about `readResolve()` as well as other serialization details at Sun's Web site dedicated to the Java Object Serialization Specification (see Resources).
One last gotcha to watch out for is the case of transient variables. If a variable is marked as transient, then it will not be serialized, and therefore it and its graph will not be copied. Instead, the value of the transient variable in the new object will be the Java language defaults (null, false, and zero). There will be no compiletime or runtime errors, which can result in behavior that is hard to debug. Just being aware of this can save a lot of time.

The deep copy technique can save a programmer many hours of work but can cause the problems described above. As always, be sure to weigh the advantages and disadvantages before deciding which method to use.

**Conclusion**
Implementing deep copy of a complex object graph can be a difficult task. The technique shown above is a simple alternative to the conventional procedure of overwriting the clone() method for every object in the graph.