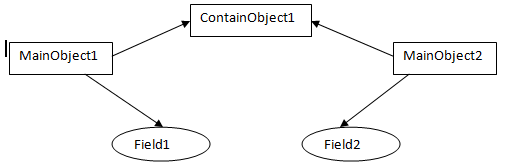
<http://www.jusfortechies.com/java/core-java/deepcopy_and_shallowcopy.php>

# Deep Copy And Shallow Copy

*What is Shallow Copy?*

Shallow copy is a bit-wise copy of an object. A new object is created that has an exact copy of the values in the original object. If any of the fields of the object are references to other objects, just the reference addresses are copied i.e., only the memory address is copied.

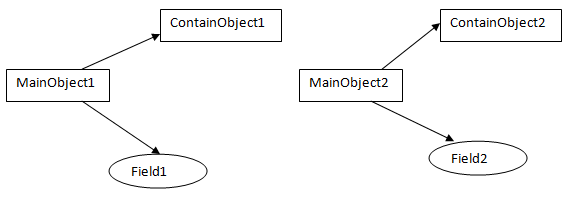


In this figure, the MainObject1 have fields "field1" of int type, and "ContainObject1" of ContainObject type. When you do a shallow copy of MainObject1, MainObject2 is created with "field3" containing the copied value of "field1" and still pointing to ContainObject1 itself. Observe here and you will find that since field1 is of primitive type, the values of it are copied to field3 but ContainedObject1 is an object, so MainObject2 is still pointing to ContainObject1. So any changes made to ContainObject1 in MainObject1 will reflect in MainObject2.

Now if this is shallow copy, lets see what's deep copy?

*What is Deep Copy?*

A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.



In this figure, the MainObject1 have fields "field1" of int type, and "ContainObject1" of ContainObject type. When you do a deep copy of MainObject1, MainObject2 is created with "field3" containing the copied value of "field1" and "ContainObject2" containing the copied value of ContainObject1.So any changes made to ContainObject1 in MainObject1 will not reflect in MainObject2.

Well, here we are with what shallow copy and deep copy are and obviously the difference between them. Now lets see how to implement them in java.

*How to implement shallow copy in java?*

Here is an example of Shallow Copy implementation

class Subject {

private String name;

public String getName() {

return name;

}

public void setName(String s) {

name = s;

}

public Subject(String s) {

name = s;

}

}

class Student implements Cloneable {

//Contained object

private Subject subj;

private String name;

public Subject getSubj() {

return subj;

}

public String getName() {

return name;

}

public void setName(String s) {

name = s;

}

public Student(String s, String sub) {

name = s;

subj = new Subject(sub);

}

public Object clone() {

//shallow copy

try {

return super.clone();

} catch (CloneNotSupportedException e) {

return null;

}

}

}

public class CopyTest {

public static void main(String[] args) {

//Original Object

Student stud = new Student("John", "Algebra");

System.out.println("Original Object: " + stud.getName() + " - "

+ stud.getSubj().getName());

//Clone Object

Student clonedStud = (Student) stud.clone();

System.out.println("Cloned Object: " + clonedStud.getName() + " - "

+ clonedStud.getSubj().getName());

stud.setName("Dan");

stud.getSubj().setName("Physics");

System.out.println("Original Object after it is updated: "

+ stud.getName() + " - " + stud.getSubj().getName());

System.out.println("Cloned Object after updating original object: "

+ clonedStud.getName() + " - " + clonedStud.getSubj().getName());

}

}

Output is:

Original Object: John - Algebra

Cloned Object: John - Algebra

Original Object after it is updated: Dan - Physics

Cloned Object after updating original object: John - Physics

In this example, all I did is, implement the class that you want to copy with Clonable interface and override clone() method of Object class and call super.clone() in it. If you observe, the changes made to "name" field of original object (Student class) is not reflected in cloned object but the changes made to "name" field of contained object (Subject class) is reflected in cloned object. This is because the cloned object carries the memory address of the Subject object but not the actual values. Hence any updates on the Subject object in Original object will reflect in Cloned object.

*How to implement deep copy in java?*

Here is an example of Deep Copy implementation. This is the same example of Shallow Copy implementation and hence I didnt write the Subject and CopyTest classes as there is no change in them.

class Student implements Cloneable {

//Contained object

private Subject subj;

private String name;

public Subject getSubj() {

return subj;

}

public String getName() {

return name;

}

public void setName(String s) {

name = s;

}

public Student(String s, String sub) {

name = s;

subj = new Subject(sub);

}

public Object clone() {

//Deep copy

Student s = new Student(name, subj.getName());

return s;

}

}

Output is:

Original Object: John - Algebra

Cloned Object: John - Algebra

Original Object after it is updated: Dan - Physics

Cloned Object after updating original object: John - Algebra

Well, if you observe here in the "Student" class, you will see only the change in the "clone()" method. Since its a deep copy, you need to create an object of the cloned class. Well if you have references in the Subject class, then you need to implement Cloneable interface in Subject class and override clone method in it and this goes on and on.

*There is an alternative way for deep copy.*

Yes, there is. You can do deep copy through [serialization](http://www.jusfortechies.com/java/core-java/serialization.php). What does serialization do? It writes out the whole object graph into a persistant store and read it back when needed, which means you will get a copy of the whole object graph whne you read it back. This is exactly what you want when you deep copy an object. Note, when you deep copy through serialization, you should make sure that all classes in the object's graph are serializable. Let me explain you this alternative way with an example. If you want to know about serialization first, check it out [here](http://www.jusfortechies.com/java/core-java/serialization.php).

public class ColoredCircle implements Serializable

{

private int x;

private int y;

public ColoredCircle(int x, int y){

this.x = x;

this.y = y;

}

public int getX(){

return x;

}

public void setX(int x){

this.x = x;

}

public int getY(){

return y;

}

public void setX(int x){

this.x = x;

}

}

public class DeepCopy

{

static public void main(String[] args)

{

ObjectOutputStream oos = null;

ObjectInputStream ois = null;

try

{

// create original serializable object

ColoredCircle c1 = new ColoredCircle(100,100);

// print it

System.out.println("Original = " + c1);

ColoredCircle c2 = null;

// deep copy

ByteArrayOutputStream bos = new ByteArrayOutputStream();

oos = new ObjectOutputStream(bos);

// serialize and pass the object

oos.writeObject(c1);

oos.flush();

ByteArrayInputStream bin =

new ByteArrayInputStream(bos.toByteArray());

ois = new ObjectInputStream(bin);

// return the new object

c2 = ois.readObject();

// verify it is the same

System.out.println("Copied = " + c2);

// change the original object's contents

c1.setX(200);

c1.setY(200);

// see what is in each one now

System.out.println("Original = " + c1);

System.out.println("Copied = " + c2);

}

catch(Exception e)

{

System.out.println("Exception in main = " + e);

}

finally

{

oos.close();

ois.close();

}

}

}

The output is:

Original = x=100,y=100

Copied = x=100,y=100

Original = x=200,y=200

Copied = x=100,y=100

All you need to do here is:

* Ensure that all classes in the object's graph are serializable.
* Create input and output streams.
* Use the input and output streams to create object input and object output streams.
* Pass the object that you want to copy to the object output stream.
* Read the new object from the object input stream and cast it back to the class of the object you sent.

In this example, I have created a ColoredCircle object, c1 and then serialized it (write it out to ByteArrayOutputStream). Then I deserialed the serialized object and saved it in c2. Later I modified the original object, c1. Then if you see the result, c1 is different from c2. c2 is deep copy of first version of c1. So its just a copy and not a reference. Now any modifications to c1 wont affect c2, the deep copy of first version of c1.

*Well this approach has got its own limitations and issues:*

As you cannot serialize a transient variable, using this approach you cannot copy the transient variables.   
Another issue is dealing with the case of a class whose object's 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.  
Next one is the performance issue. Creating a socket, serializing an object, passing it through the socket, and then deserializing it is slow compared to calling methods in existing objects. I say, there will be vast difference in the performance. If your code is performance critical, I suggest dont go for this approach. It takes almost 100 times more time to deep copy the object than the way you do by implementing Clonable interface.

*When to do shallow copy and deep copy?*

Its very simple that if the object has only primitive fields, then obviously you will go for shallow copy but if the object has references to other objects, then based on the requiement, shallow copy or deep copy should be chosen. What I mean here is, if the references are not modified anytime, then there is no point in going for deep copy. You can just opt shallow copy. But if the references are modified often, then you need to go for deep copy. Again there is no hard and fast rule, it all depends on the requirement.

*Finally lets have a word about rarely used option - Lazy copy*

A lazy copy is a combination of both shallow copy and deep copy. When initially copying an object, a (fast) shallow copy is used. A counter is also used to track how many objects share the data. When the program wants to modify the original object, it can determine if the data is shared (by examining the counter) and can do a deep copy at that time if necessary.

Lazy copy looks to the outside just as a deep copy but takes advantage of the speed of a shallow copy whenever possible. It can be used when the references in the original object are not modified often. The downside are rather high but constant base costs because of the counter. Also, in certain situations, circular references can also cause problems.

|  |  |
| --- | --- |
| up vote93down voteaccepted | A safe way is to serialize the object, then deserialize. This ensures everything is a brand new reference.  [Here's an article](http://javatechniques.com/blog/faster-deep-copies-of-java-objects/) about how to do this efficiently.  Caveats: It's possible for classes to override serialization such that new instances are not created, e.g. for singletons. Also this of course doesn't work if your classes aren't Serializable. |

## **Faster Deep Copies of Java Objects (**[single-threaded approach](http://javatechniques.com/blog/faster-deep-copies-of-java-objects/).)

The **java.lang.Object** root superclass defines a **clone()** method that will, assuming the subclass implements the **java.lang.Cloneable** interface, return a copy of the object. While Java classes are free to override this method to do more complex kinds of cloning, the default behavior of **clone()** is to return a *shallow* copy of the object. This means that the values of all of the origical object’s fields are copied to the fields of the new object.

A property of shallow copies is that fields that refer to other objects will point to *the same* objects in both the original and the clone. For fields that contain primitive or immutable values (**int**, **String**,**float**, etc…), there is little chance of this causing problems. For mutable objects, however, cloning can lead to unexpected results. Figure 1 shows an example.

import java.util.Vector;

public class Example1 {

public static void main(String[] args) {

// Make a Vector

Vector original = new Vector();

// Make a StringBuffer and add it to the Vector

StringBuffer text = new StringBuffer("The quick brown fox");

original.addElement(text);

// Clone the vector and print out the contents

Vector clone = (Vector) original.clone();

System.out.println("A. After cloning");

printVectorContents(original, "original");

printVectorContents(clone, "clone");

System.out.println(

"--------------------------------------------------------");

System.out.println();

// Add another object (an Integer) to the clone and

// print out the contents

clone.addElement(new Integer(5));

System.out.println("B. After adding an Integer to the clone");

printVectorContents(original, "original");

printVectorContents(clone, "clone");

System.out.println(

"--------------------------------------------------------");

System.out.println();

// Change the StringBuffer contents

text.append(" jumps over the lazy dog.");

System.out.println("C. After modifying one of original's elements");

printVectorContents(original, "original");

printVectorContents(clone, "clone");

System.out.println(

"--------------------------------------------------------");

System.out.println();

}

public static void printVectorContents(Vector v, String name) {

System.out.println(" Contents of \"" + name + "\":");

// For each element in the vector, print out the index, the

// class of the element, and the element itself

for (int i = 0; i < v.size(); i++) {

Object element = v.elementAt(i);

System.out.println(" " + i + " (" +

element.getClass().getName() + "): " +

element);

}

System.out.println();

}

}

Figure 1. Modifying **Vector** contents after cloning

In this example we create a **Vector** and add a **StringBuffer** to it. Note that **StringBuffer** (unlike, for example, **String** is mutable — it’s contents can be changed after creation. Figure 2 shows the output of the example in Figure 1.

> java Example1

A. After cloning

Contents of "original":

0 (java.lang.StringBuffer): The quick brown fox

Contents of "clone":

0 (java.lang.StringBuffer): The quick brown fox

--------------------------------------------------------

B. After adding an Integer to the clone

Contents of "original":

0 (java.lang.StringBuffer): The quick brown fox

Contents of "clone":

0 (java.lang.StringBuffer): The quick brown fox

1 (java.lang.Integer): 5

--------------------------------------------------------

C. After modifying one of original's elements

Contents of "original":

0 (java.lang.StringBuffer): The quick brown fox jumps over the lazy dog.

Contents of "clone":

0 (java.lang.StringBuffer): The quick brown fox jumps over the lazy dog.

1 (java.lang.Integer): 5

--------------------------------------------------------

Figure 2. Output from the example code in Figure 1

In the first block of output (“A”), we see that the clone operation was successful: The original vector and the clone have the same size (1), content types, and values. The second block of output (“B”) shows that the original vector and its clone are distinct objects. If we add another element to the clone, it only appears in the clone, and not in the original. The third block of output (“C”) is, however, a little trickier. Modifying the **StringBuffer** that was added to the original vector has changed the value of the first element of *both* the original vector and its clone. The explanation for this lies in the fact that **clone** made a shallow copy of the vector, so both vectors now point to the exact same**StringBuffer** instance.

This is, of course, sometimes exactly the behavior that you need. In other cases, however, it can lead to frustrating and inexplicable errors, as the state of an object seems to change “behind your back”.

The solution to this problem is to make a *deep copy* of the object. A deep copy makes a distinct copy of each of the object’s fields, recursing through the entire graph of other objects referenced by the object being copied. The Java API provides no deep-copy equivalent to **Object.clone()**. One solution is to simply implement your own custom method (e.g., **deepCopy()**) that returns a deep copy of an instance of one of your classes. This may be the best solution if you need a complex mixture of deep and shallow copies for different fields, but has a few significant drawbacks:

1. You must be able to modify the class (i.e., have the source code) or implement a subclass. If you have a third-party class for which you do not have the source and which is marked **final**, you are out of luck.
2. You must be able to access all of the fields of the class’s superclasses. If significant parts of the object’s state are contained in **private** fields of a superclass, you will not be able to access them.
3. You must have a way to make copies of instances of all of the other kinds of objects that the object references. This is particularly problematic if the exact classes of referenced objects cannot be known until runtime.
4. Custom deep copy methods are tedious to implement, easy to get wrong, and difficult to maintain. The method must be revisited any time a change is made to the class or to any of its superclasses.

A common solution to the deep copy problem is to use Java Object Serialization (JOS). The idea is simple: Write the object to an array using JOS’s **ObjectOutputStream** and then use**ObjectInputStream** to reconsistute a copy of the object. The result will be a completely distinct object, with completely distinct referenced objects. JOS takes care of all of the details: superclass fields, following object graphs, and handling repeated references to the same object within the graph. Figure 3 shows a first draft of a utility class that uses JOS for making deep copies.

import java.io.IOException;

import java.io.ByteArrayInputStream;

import java.io.ByteArrayOutputStream;

import java.io.ObjectOutputStream;

import java.io.ObjectInputStream;

/\*\*

\* Utility for making deep copies (vs. clone()'s shallow copies) of

\* objects. Objects are first serialized and then deserialized. Error

\* checking is fairly minimal in this implementation. If an object is

\* encountered that cannot be serialized (or that references an object

\* that cannot be serialized) an error is printed to System.err and

\* null is returned. Depending on your specific application, it might

\* make more sense to have copy(...) re-throw the exception.

\*

\* A later version of this class includes some minor optimizations.

\*/

public class UnoptimizedDeepCopy {

/\*\*

\* Returns a copy of the object, or null if the object cannot

\* be serialized.

\*/

public static Object copy(Object orig) {

Object obj = null;

try {

// Write the object out to a byte array

ByteArrayOutputStream bos = new ByteArrayOutputStream();

ObjectOutputStream out = new ObjectOutputStream(bos);

out.writeObject(orig);

out.flush();

out.close();

// Make an input stream from the byte array and read

// a copy of the object back in.

ObjectInputStream in = new ObjectInputStream(

new ByteArrayInputStream(bos.toByteArray()));

obj = in.readObject();

}

catch(IOException e) {

e.printStackTrace();

}

catch(ClassNotFoundException cnfe) {

cnfe.printStackTrace();

}

return obj;

}

}

Figure 3. Using Java Object Serialization to make a deep copy

Unfortunately, this approach has some problems, too:

1. It will only work when the object being copied, as well as all of the other objects references directly or indirectly by the object, are serializable. (In other words, they must implement**java.io.Serializable**.) Fortunately it is often sufficient to simply declare that a given class**implements java.io.Serializable** and let Java’s default serialization mechanisms do their thing.
2. Java Object Serialization is slow, and using it to make a deep copy requires both serializing and deserializing. There are ways to speed it up (e.g., by pre-computing serial version ids and defining custom **readObject()** and **writeObject()** methods), but this will usually be the primary bottleneck.
3. The byte array stream implementations included in the **java.io** package are designed to be general enough to perform reasonable well for data of different sizes and to be safe to use in a multi-threaded environment. These characteristics, however, slow down**ByteArrayOutputStream** and (to a lesser extent) **ByteArrayInputStream**.

The first two of these problems cannot be addressed in a general way. We can, however, use alternative implementations of **ByteArrayOutputStream** and **ByteArrayInputStream** that makes three simple optimizations:

1. **ByteArrayOutputStream**, by default, begins with a 32 byte array for the output. As content is written to the stream, the required size of the content is computed and (if necessary), the array is expanded to the greater of the required size or twice the current size. JOS produces output that is somewhat bloated (for example, fully qualifies path names are included in uncompressed string form), so the 32 byte default starting size means that lots of small arrays are created, copied into, and thrown away as data is written. This has an easy fix: construct the array with a larger inital size.
2. All of the methods of **ByteArrayOutputStream** that modify the contents of the byte array are**synchronized**. In general this is a good idea, but in this case we can be certain that only a single thread will ever be accessing the stream. Removing the synchronization will speed things up a little. **ByteArrayInputStream**‘s methods are also synchronized.
3. The **toByteArray()** method creates and returns a *copy* of the stream’s byte array. Again, this is usually a good idea: If you retrieve the byte array and then continue writing to the stream, the retrieved byte array should not change. For this case, however, creating another byte array and copying into it merely wastes cycles and makes extra work for the garbage collector.

An optimized implementation of **ByteArrayOutputStream** is shown in Figure 4.

import java.io.OutputStream;

import java.io.IOException;

import java.io.InputStream;

import java.io.ByteArrayInputStream;

/\*\*

\* ByteArrayOutputStream implementation that doesn't synchronize methods

\* and doesn't copy the data on toByteArray().

\*/

public class FastByteArrayOutputStream extends OutputStream {

/\*\*

\* Buffer and size

\*/

protected byte[] buf = null;

protected int size = 0;

/\*\*

\* Constructs a stream with buffer capacity size 5K

\*/

public FastByteArrayOutputStream() {

this(5 \* 1024);

}

/\*\*

\* Constructs a stream with the given initial size

\*/

public FastByteArrayOutputStream(int initSize) {

this.size = 0;

this.buf = new byte[initSize];

}

/\*\*

\* Ensures that we have a large enough buffer for the given size.

\*/

private void verifyBufferSize(int sz) {

if (sz > buf.length) {

byte[] old = buf;

buf = new byte[Math.max(sz, 2 \* buf.length )];

System.arraycopy(old, 0, buf, 0, old.length);

old = null;

}

}

public int getSize() {

return size;

}

/\*\*

\* Returns the byte array containing the written data. Note that this

\* array will almost always be larger than the amount of data actually

\* written.

\*/

public byte[] getByteArray() {

return buf;

}

public final void write(byte b[]) {

verifyBufferSize(size + b.length);

System.arraycopy(b, 0, buf, size, b.length);

size += b.length;

}

public final void write(byte b[], int off, int len) {

verifyBufferSize(size + len);

System.arraycopy(b, off, buf, size, len);

size += len;

}

public final void write(int b) {

verifyBufferSize(size + 1);

buf[size++] = (byte) b;

}

public void reset() {

size = 0;

}

/\*\*

\* Returns a ByteArrayInputStream for reading back the written data

\*/

public InputStream getInputStream() {

return new FastByteArrayInputStream(buf, size);

}

}

Figure 4. Optimized version of **ByteArrayOutputStream**

The **getInputStream()** method returns an instance of an optimized version of**ByteArrayInputStream** that has unsychronized methods. The implementation of**FastByteArrayInputStream** is shown in Figure 5.

import java.io.InputStream;

import java.io.IOException;

/\*\*

\* ByteArrayInputStream implementation that does not synchronize methods.

\*/

public class FastByteArrayInputStream extends InputStream {

/\*\*

\* Our byte buffer

\*/

protected byte[] buf = null;

/\*\*

\* Number of bytes that we can read from the buffer

\*/

protected int count = 0;

/\*\*

\* Number of bytes that have been read from the buffer

\*/

protected int pos = 0;

public FastByteArrayInputStream(byte[] buf, int count) {

this.buf = buf;

this.count = count;

}

public final int available() {

return count - pos;

}

public final int read() {

return (pos < count) ? (buf[pos++] & 0xff) : -1;

}

public final int read(byte[] b, int off, int len) {

if (pos >= count)

return -1;

if ((pos + len) > count)

len = (count - pos);

System.arraycopy(buf, pos, b, off, len);

pos += len;

return len;

}

public final long skip(long n) {

if ((pos + n) > count)

n = count - pos;

if (n < 0)

return 0;

pos += n;

return n;

}

}

Figure 5. Optimized version of **ByteArrayInputStream**.

Figure 6 shows a version of a deep copy utility that uses these classes:

import java.io.IOException;

import java.io.ByteArrayInputStream;

import java.io.ByteArrayOutputStream;

import java.io.ObjectOutputStream;

import java.io.ObjectInputStream;

/\*\*

\* Utility for making deep copies (vs. clone()'s shallow copies) of

\* objects. Objects are first serialized and then deserialized. Error

\* checking is fairly minimal in this implementation. If an object is

\* encountered that cannot be serialized (or that references an object

\* that cannot be serialized) an error is printed to System.err and

\* null is returned. Depending on your specific application, it might

\* make more sense to have copy(...) re-throw the exception.

\*/

public class DeepCopy {

/\*\*

\* Returns a copy of the object, or null if the object cannot

\* be serialized.

\*/

public static Object copy(Object orig) {

Object obj = null;

try {

// Write the object out to a byte array

FastByteArrayOutputStream fbos =

new FastByteArrayOutputStream();

ObjectOutputStream out = new ObjectOutputStream(fbos);

out.writeObject(orig);

out.flush();

out.close();

// Retrieve an input stream from the byte array and read

// a copy of the object back in.

ObjectInputStream in =

new ObjectInputStream(fbos.getInputStream());

obj = in.readObject();

}

catch(IOException e) {

e.printStackTrace();

}

catch(ClassNotFoundException cnfe) {

cnfe.printStackTrace();

}

return obj;

}

}

Figure 6. Deep-copy implementation using optimized byte array streams

The extent of the speed boost will depend on a number of factors in your specific application (more on this later), but the simple class shown in Figure 7 tests the optimized and unoptimized versions of the deep copy utility by repeatedly copying a large object.

import java.util.Hashtable;

import java.util.Vector;

import java.util.Date;

public class SpeedTest {

public static void main(String[] args) {

// Make a reasonable large test object. Note that this doesn't

// do anything useful -- it is simply intended to be large, have

// several levels of references, and be somewhat random. We start

// with a hashtable and add vectors to it, where each element in

// the vector is a Date object (initialized to the current time),

// a semi-random string, and a (circular) reference back to the

// object itself. In this case the resulting object produces

// a serialized representation that is approximate 700K.

Hashtable obj = new Hashtable();

for (int i = 0; i < 100; i++) {

Vector v = new Vector();

for (int j = 0; j < 100; j++) {

v.addElement(new Object[] {

new Date(),

"A random number: " + Math.random(),

obj

});

}

obj.put(new Integer(i), v);

}

int iterations = 10;

// Make copies of the object using the unoptimized version

// of the deep copy utility.

long unoptimizedTime = 0L;

for (int i = 0; i < iterations; i++) {

long start = System.currentTimeMillis();

Object copy = UnoptimizedDeepCopy.copy(obj);

unoptimizedTime += (System.currentTimeMillis() - start);

// Avoid having GC run while we are timing...

copy = null;

System.gc();

}

// Repeat with the optimized version

long optimizedTime = 0L;

for (int i = 0; i < iterations; i++) {

long start = System.currentTimeMillis();

Object copy = DeepCopy.copy(obj);

optimizedTime += (System.currentTimeMillis() - start);

// Avoid having GC run while we are timing...

copy = null;

System.gc();

}

System.out.println("Unoptimized time: " + unoptimizedTime);

System.out.println(" Optimized time: " + optimizedTime);

}

}

Figure 7. Testing the two deep copy implementations.

A few notes about this test:

* The object that we are copying is large. While somewhat random, it will generally have a serialized size of around 700 Kbytes.
* The most significant speed boost comes from avoid extra copying of data in**FastByteArray*Output*Stream**. This has several implications:
  1. Using the unsynchronized **FastByteArrayInputStream** speeds things up a little, but the standard **java.io.ByteArrayInputStream** is nearly as fast.
  2. Performance is mildly sensitive to the initial buffer size in **FastByteArrayOutputStream**, but is much more sensitive to the rate at which the buffer grows. If the objects you are copying tend to be of similar size, copying will be much faster if you initialize the buffer size and tweak the rate of growth.
* Measuring speed using elapsed time between two calls to **System.currentTimeMillis()** is problematic, but for single-threaded applications and testing relatively slow operations it is sufficient. A number of commercial tools (such as JProfiler) will give more accurate per-method timing data.
* Testing code in a loop is also problematic, since the first few iterations will be slower until HotSpot decides to compile the code. Testing larger numbers of iterations aleviates this problems.
* Garbage collection further complicates matters, particularly in cases where lots of memory is allocated. In this example, we manually invoke the garbage collector after each copy to try to keep it from running while a copy is in progress.

These caveats aside, the performance difference is sigificant. For example, the code as shown in Figure 7 (on a 500Mhz G3 Macintosh iBook running OSX 10.3 and Java 1.4.1) reveals that the unoptimized version requires about 1.8 seconds per copy, while the optimized version only requires about 1.3 seconds. Whether or not this difference is signficant will, of course, depend on the frequency with which your application does deep copies and the size of the objects being copied.

For very large objects, an extension to this approach can reduce the peak memory footprint by serializing and deserializing in parallel threads. See "[Low-Memory Deep Copy Technique for Java Objects](http://javatechniques.com/blog/low-memory-deep-copy-technique-for-java-objects/)" for more information.

## **Low-Memory Deep Copy Technique for Java Objects**

“[Faster Deep Copies of Java Objects](http://javatechniques.com/blog/faster-deep-copies-of-java-objects/)” explains the concept of “deep copies” of Java objects and illustrates a common approach for copying objects using Java Object Serialization. A downside of the illustrated approach is that it requires creation of a byte array containing the entire serialized representation of the object being copied. Creation of this array is inherently wasteful, as it is thrown away once the copy has been made.

It would, of course, be possible to save the array and re-use it for subsequent object copies, though that may not be appropriate in some applications. An alternate approach is to use**java.io.PipedInputStream** and **java.io.PipedOutputStream** to serialize the object in one thread and simultaneously deserialize it in another thread. This technique is more complex, but allows the two halves of the operation to coordinate using a small buffer rather than a large byte array. Figure 1 shows an example copy utility that uses this approach.

import java.io.IOException;

import java.io.PipedOutputStream;

import java.io.PipedInputStream;

import java.io.ObjectOutputStream;

import java.io.ObjectInputStream;

/\*\*

\* Utility for making deep copies (vs. clone()'s shallow copies) of objects

\* in a memory efficient way. Objects are serialized in the calling thread and

\* de-serialized in another thread.

\*

\* Error checking is fairly minimal in this implementation. If an object is

\* encountered that cannot be serialized (or that references an object

\* that cannot be serialized) an error is printed to System.err and

\* null is returned. Depending on your specific application, it might

\* make more sense to have copy(...) re-throw the exception.

\*/

public class PipedDeepCopy {

/\*\*

\* Flag object used internally to indicate that deserialization failed.

\*/

private static final Object ERROR = new Object();

/\*\*

\* Returns a copy of the object, or null if the object cannot

\* be serialized.

\*/

public static Object copy(Object orig) {

Object obj = null;

try {

// Make a connected pair of piped streams

PipedInputStream in = new PipedInputStream();

PipedOutputStream pos = new PipedOutputStream(in);

// Make a deserializer thread (see inner class below)

Deserializer des = new Deserializer(in);

// Write the object to the pipe

ObjectOutputStream out = new ObjectOutputStream(pos);

out.writeObject(orig);

// Wait for the object to be deserialized

obj = des.getDeserializedObject();

// See if something went wrong

if (obj == ERROR)

obj = null;

}

catch(IOException ioe) {

ioe.printStackTrace();

}

return obj;

}

/\*\*

\* Thread subclass that handles deserializing from a PipedInputStream.

\*/

private static class Deserializer extends Thread {

/\*\*

\* Object that we are deserializing

\*/

private Object obj = null;

/\*\*

\* Lock that we block on while deserialization is happening

\*/

private Object lock = null;

/\*\*

\* InputStream that the object is deserialized from.

\*/

private PipedInputStream in = null;

public Deserializer(PipedInputStream pin) throws IOException {

lock = new Object();

this.in = pin;

start();

}

public void run() {

Object o = null;

try {

ObjectInputStream oin = new ObjectInputStream(in);

o = oin.readObject();

}

catch(IOException e) {

// This should never happen. If it does we make sure

// that a the object is set to a flag that indicates

// deserialization was not possible.

e.printStackTrace();

}

catch(ClassNotFoundException cnfe) {

// Same here...

cnfe.printStackTrace();

}

synchronized(lock) {

if (o == null)

obj = ERROR;

else

obj = o;

lock.notifyAll();

}

}

/\*\*

\* Returns the deserialized object. This method will block until

\* the object is actually available.

\*/

public Object getDeserializedObject() {

// Wait for the object to show up

try {

synchronized(lock) {

while (obj == null) {

lock.wait();

}

}

}

catch(InterruptedException ie) {

// If we are interrupted we just return null

}

return obj;

}

}

}

Figure 1. Deep copy utility using **PipedInputStream** and **PipedOutputStream**

The thread that calls **copy(…)** creates two connected piped streams and a separate thread to handle deserialization. The object is written to one side of the pipe and read (in the **Deserializer** thread) from the other. Given the non-deterministic nature of thread scheduling, there is no guarantee that the **Deserializer** thread will finish reading immediately after the calling thread has finished writing, so the calling thread will block if necessary until reading has finished.

Figure 2 shows an example of copying a large object using the **PipedDeepCopy** class shown in Figure 1.

import java.util.Hashtable;

import java.util.Vector;

import java.util.Date;

public class Example1 {

public static void main(String[] args) {

// Make a reasonable large test object. Note that this doesn't

// do anything useful -- it is simply intended to be large, have

// several levels of references, and be somewhat random. We start

// with a hashtable and add vectors to it, where each element in

// the vector is a Date object (initialized to the current time),

// a semi-random string, and a (circular) reference back to the

// object itself. In this case the resulting object produces

// a serialized representation that is approximate 700K.

Hashtable obj = new Hashtable();

for (int i = 0; i < 100; i++) {

Vector v = new Vector();

for (int j = 0; j < 100; j++) {

v.addElement(new Object[] {

new Date(),

"A random number: " + Math.random(),

obj

});

}

obj.put(new Integer(i), v);

}

int iterations = 10;

// Run piped version

long time = 0L;

for (int i = 0; i < iterations; i++) {

long start = System.currentTimeMillis();

Object copy = PipedDeepCopy.copy(obj);

time += (System.currentTimeMillis() - start);

// Avoid having GC run while we are timing...

copy = null;

System.gc();

}

System.out.println("Piped time: " + time);

}

}

Figure 2. Example of copying a large object

The good news is that the extra memory needed to make the copy is essentially bounded by the size of the buffer used by **PipedInputStream**. By default, this buffer is 1024 bytes. The bad news is that this approach is more than twice as slow as even the unoptimized variant of the deep copy utility illustrated in “[Faster Deep Copies of Java Objects](http://javatechniques.com/public/java/docs/basics/faster-deep-copy.html)“. The reason for this is the added overhead of coordination between the two threads. In the example in Figure 2, copying the 700K object requires that the buffer in **PipedInputStream** be read and overwritten 700 times. The two threads must do a fair bit of handshaking, with the **ObjectInputStream** waiting for the **ObjectOutputStream** to write needed data to the buffer, or the **ObjectOutputStream** waiting for the **ObjectInputStream** to pull data from the buffer so that more can be written.

Increasing the size of the buffer used by **PipedInputStream** can help. Unfortunately,**PipedInputStream** does not provide a way to change the buffer size. It can, however, be easily subclassed to provide this capability. Figure 3 illustrates this.

import java.io.PipedInputStream;

/\*\*

\* PipedInputStream subclass that allows buffer size to be set to

\* a value larger than the default 1024 bytes.

\*/

private static class AdjustableBufferPipedInputStream extends

PipedInputStream {

public AdjustableBufferPipedInputStream(int bufSize) {

super();

buffer = new byte[bufSize];

}

}

Figure 3. **PipedInputStream** subclass that allows buffer size to be specified in the constructor.

This helps some, but the overhead of thread coordination is still a limiting factor. If, for example, the buffer size is set to a value larger than the serialized size of the object being copied (effectively eliminating the memory advantages of the piped approach), the copy still takes nearly 50% longer than the [single-threaded approach](http://javatechniques.com/blog/faster-deep-copies-of-java-objects/).