

# On [Transient], ObjectUtil.copy(), and Casting

By darron on August 13, 2007 10:20 AM | [1 Comment](#) | [No TrackBacks](#)

Source URL: <http://www.darronschall.com/weblog/2007/08/on-transient-objectutilcopy-and-casting.cfm>

Accessed: 01/08/2010

I ran into an interesting situation not too long ago where [ObjectUtil.copy](#) wasn't working quite as I had expected. The solution that I came up with was to rely on [Transient] metadata. Confused? Let me explain...

ObjectUtil.copy is a wonderful utility method to perform a [deep copy](#) of an object. Its implementation is also amazingly simple:

```
1. var buffer:ByteArray = new ByteArray();
2. buffer.writeObject(value);
3. buffer.position = 0;
4. var result:Object = buffer.readObject();
5. return result;
```

Rather than using introspection and recursively copying properties from one object to another, the entire object is serialized into a array of bytes. When the bytes are deserialized, a brand new object is created copying all of the original contents.

Simple, but effective. Instead of writing a custom deep copy algorithm, the copy() method uses the built in Flash Player AMF capabilities. This is the same serialization that Flash Remoting uses. When you send an object to the server via Flash Remoting (<mx:RemoteObject/>), you send an AMF array of bytes describing the object that the server deserializes to create an identical object on the server.

Because of using AMF behind the scenes to perform object copies, a few interesting things happen:

- If you try to cast the results of a copy to a class, the cast might fail.
- Using the [Transient] metadata tag affects the output of copy().

When an object is deserialized from AMF, it does not automatically get created as a class instance. The object might have all of the properties of a class instance, but it will not be a true class instance unless the AMF packet includes type information about the object. The type information gets added to AMF in one of two ways:

- Using the [\[RemoteClass\] metadata](#) tag.
- Using the [registerClassAlias\(\)](#) method.

If you're using [RemoteClass] metadata on a class instance being copied, then it is safe to cast the result as that particular class instance:

```
1. // The works if Book has [RemoteClass] metadata
2. var bookCopy:Book = Book( ObjectUtil.copy( book ) );
```

If you're not using [RemoteClass], before performing the copy you need to register the class against a string alias. The alias is written to the AMF packet so that when the object is deserialized, it can be created as the proper type. For example:

```
1. // Book doesn't have [RemoteClass] metadata, so associate the my.package.Book string
2. // with a reference to the Book class.
3. registerClassAlias( "my.package.Book", Book );
4.
5. // Now we can cast the result of the copy without errors.
6. var bookCopy:Book = Book( ObjectUtil.copy( book ) );
```

So, what about [Transient], and how is this related to the issue I ran into where the copy wasn't working as I had expected? Now that you know how copy works behind the scenes, let's talk about [Transient].

The [Transient] metadata tag is not very well documented in Flex 2 or 2.0.1. If you look in some of the LiveCycle Data Services documentation you can see mentions of it, but there isn't a dedicated page on the subject. In the beta Flex 3 documentation, [\[Transient\] is finally explained](#). There is also some documentation gathered on [The Flex Non-Docs weblog](#).

Essentially, what it comes to is that **using [Transient] on a property removes that property from the AMF packet during serialization**. This is important. The reason [Transient] properties are not sent over the wire via Flash Remoting is because, again, they're not included in the AMF packet.

With the background information explained, onto the problem I ran into. Here's a small class that doesn't copy correctly (after the jump):

```
1. package
2. {
3.
4. public class Example
5. {
6.
7.     /** Constant for when the flag value indicates option 1. */
8.     public static const OPTION_1:String = "1";
9.
10.    /** Constant for when the flag value indicates option 2. */
```

```

11. public static const OPTION_2:String = "2";
12.
13. /** Constant for when th flag vlaue indicates no option. */
14. public static const OPTION_NONE:String = "0";
15.
16. /** One of the option constants, determines what value is used for. */
17. public var flag:String;
18.
19. /** The numeric value for whatever option the flag indicates. */
20. public var value:int;
21.
22. // =====
23. // option1 property
24. // =====
25.
26. /**
27.  * Helper method to get/set the value of option 1. Returns -1 if option 1
28.  * is not the flag value.
29.  */
30. public function get option1():int
31. {
32.     return flag == OPTION_1 ? value : -1;
33. }
34.
35. public function set option1( value:int ):void
36. {
37.     // Anytime the option1 setter is called, set the flag and save the value
38.     flag = OPTION_1;
39.     this.value = value;
40. }
41.
42. // =====
43. // option2 property
44. // =====
45.
46. /**
47.  * Helper method to get/set the value of option 2. Returns -1 if option 2
48.  * is not the flag value.
49.  */
50. public function get option2():int
51. {
52.     return flag == OPTION_2 ? value : -1;
53. }
54.
55. public function set option2( value:int ):void
56. {
57.     // Anytime the option2 setter is called, set the flag and save the value
58.     flag = OPTION_2;
59.     this.value = value;
60. }
61.
62. } // end class
63. } // end package

```

What's important about this class is that there are really only two values that we care about - *flag*, and *value*. I'm using option1 and option2 as helper getter/setters to read and manipulate those properties.

Trying to copy this class yields some interesting behavior:

```

1. // Create an object and assign a value
2. var example:Example = new Example();
3. example.option2 = 123;
4.
5. // Register the class so deserialization preserves the type information
6. registerClassAlias( "Example", Example );
7.
8. // Make a copy of the original object
9. var copy:Example = Example( ObjectUtil.copy( example ) );
10.
11. trace( "example: " + example.option2 ); // example: 123
12. trace( "copy: " + copy.option2 ); // copy: -1
13.
14. trace( example.flag ); // 2
15. trace( copy.flag ); // 1

```

Since *example* and *copy* have different option2 values, obviously the copy didn't complete as expected. Where did we go wrong?

When the `copy()` method serialized the data as AMF, Flash Player not only wrote the `flag` and `value` properties and their values, but it *also* wrote the `option1` and `option2` values. When the deserialization happened, all 4 properties were read back in. The `option1` property was read in *after* `option2`. This means the `option1` setter was called and changed the values of both `flag` and `value` incorrectly. Oops!

The solution, now that we know how `[Transient]` relates to AMF, is to tag the `option1` and `option2` helper properties as transient. This excludes them from the AMF serialization and prevents errors when the setters would be called during deserialization:

```
1. // Flag option1 as transient so it is not included as part
2. // of the object when copied or sent to the server
3. [Transient]
4. public function get option1():int
5. // ...
6. // ...likewise with option2
```

After adding `[Transient]`, when the example code is run again only *flag* and *value* are serialized (and therefore deserialized and copied). This corrects the original behavior and the copy's `option2` value is the same as example's value.

I hope this helps in your understanding of AMF, how it relates to class instances and casting, and what kind of effect the `[Transient]` metadata actually has behind the scenes. Even if a class is never meant to be sent to the server, `[Transient]` can still be useful.