

Re: Abstract class variables question

Source:

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.csharp/2007-11/msg02315.html>

- *From:* "tshad" <tfs@xxxxxxxxxxxxxxxx>
 - *Date:* Sun, 18 Nov 2007 12:55:03 -0800
-

I think I understand boxing a little better now.

After reading your information and a couple of other articles – I think I have a better idea what it is and why it is used. The problem was the WHY. I had seen other articles on the HOW but not much on the why:

```
int i,j;
object o = i; // Boxed
j = (object) o // unboxed.
```

This shows how it is done but not why you would use it.

You're right in that I didn't understand the string type not being boxed and the values being boxed and why they were different. But now if I understand it correctly:

- 1) value types are stored on the stack (faster)
- 2) reference types are stored on the heap and have a reference/pointer to the object that is on the heap.
- 3) value types are copied to the heap and made into an object and reference type (in essence) that now has a reference/pointer pointing to the new object.
- 4) String types are already reference types and all we are doing when we do "object o = str" is create a new reference/pointer that points to the object on the heap.

In the case of a non-string object, the object is the same but we have 2 pointers to that same object. If the object changes value and both references point to the same object, they will be the same.

But in the case of a string, which is immutable, the 1st reference would point to the first string and the 2nd reference (after the string is changed) will point to the new string. And that would be the case, even if we didn't move it to a new object.

```
string A = "something";
string B;
B = A;
```

Re: Abstract class variables question

```
A += " Else";
```

gives me the same result as:

```
string s1 = "something";  
object o1 = s1;  
s1 += " Else";
```

Where A <> B and o1 <> s1 because in both cases a new string is created even though we think we are changing the strings.

I also found a better article/tutorial which I think was trying to say the same thing as the first one but this one does it better. He isn't addressing the string reference type but address reference types in general. http://www.jaggersoft.com/csharp_course/13_Boxing_files/frame.htm

I think he is saying the same thing you are saying (but he also doesn't go into the string reference type).

Hopefully I am getting it right this time.

"Peter Duniho" <NpOeStPeAdM@xxxxxxxxxxxxxxxxxxxx> wrote in message <news:2007111615315950073-NpOeStPeAdM@xxxxxxxxxxxxxxxxxxxx>

On 2007-11-16 14:24:13 -0800, "tshad" <tfs@xxxxxxxxxxxxxxxx> said:

Not really. _objCurrent is a field inside a class. All classes are reference types, which means that all classes are allocated from the heap. So _objCurrent is a memory location in the heap, that references something else that's stored in the heap.

I understand that it is a field. But I thought that since the _objCurrent is an object type that it is just a pointer but not actually an object until it is assigned to something so it is null.

I'm not really sure what that has to do with whether the variable is stored on the stack or not.

_objCurrent isn't on the stack, which is all I was pointing out.

Re: Abstract class variables question

```
class BoxedInt
{
int _value;

BoxedInt(int value)
{
_value = value;
}

public int Value
{
get { return _value; }
}
}
```

And then somewhere in code, doing this:

```
int i = 5;
BoxedInt boxed = new BoxedInt(i);
```

The current value of "i" is passed into the constructor, where it's copied to the private field "_value". You can read it back out, which then copies the value from the private field to wherever you assign it. But "i" is only relevant when the instance is first constructed, and there only as it's passed by value to the constructor.

So in this case I have "i" which is a value Type and boxed.i (if i were public) which are completely different even if they have the same value at the start. If I then change "i" to 30, boxed.i will still = 5.

That much is basically correct, not counting the hand-waving. :) There is more to boxing though than just allocating a large enough area to hold the value itself. It's oversimplifying things to say that just the value is stored (that is, a boxed int for example is going to take more space than just the 32 bits for the int).

Really?

Really. :)

Re: Abstract class variables question

Does it have more information about the actual object, such as what type of object it is?

All that information is with the object itself, not the variable that refers to it. The variable referencing the object doesn't need to store it at all, because the information can always be obtained from the instance data itself.

Because a reference is always just 32 bits. The act of boxing a value type needs to know how much space to allocate, to contain the value type along with the class that's containing it. But once the value type has been boxed (and this occurs before the assignment to the Data property), it's just a regular reference and always only takes as much space to store as any reference: 32 bits.

That is what is confusing about boxing.

According to the article above, unless I missed something, boxing actually moves the data and you are looking at a different piece of data.

Define "move". To me, "move" means you've removed something from one place, and put it somewhere else.

My mistake.

I meant copy and not move. Which is why they would be different – which is what the guy in the original article was trying to point out. That when you box the value type, the original value is still the same in the same place (on the stack in his example) and the boxed value is a DIFFERENT variable (on the heap) with the same value at the moment but since they are actually different variables, if you change one you don't affect the other.

That's definitely not what happens with boxing.

Re: Abstract class variables question

Instead, boxing makes a whole new copy of the original value type data, wraps it up in a reference type instance, and gives you the reference to that instance back.

For example:

```
string A = "something"  
string B;  
B = A
```

Both A and B are referencing the same location. Such that if I say 'A = "Something else";', A and B will both be equal to "Something Else".

Bad example, for a few reasons:

I understand now why this is a bad example.

- 1) the String type is a class, meaning it's a reference type, meaning it never gets boxed
- 2) the String type is immutable, meaning that if you write code that assigns a new value to A, the string referenced by B definitely will not change (it will still refer to "something").
- 3) In C#, you can't overload or override the assignment operator, and so an assignment to a reference type variable will always replace the reference. The example you've offered will never ever work in C#, even for a mutable class.

So, what sort of example would work? Here's one that I think gets at what you're trying to show:

```
class Mutable  
{  
    string _str;  
  
    Mutable(string str)  
    {  
        _str = str;  
    }  
  
    public string String  
    {  
        get { return _str; }  
        set { _str = value; }  
    }  
}
```

Re: Abstract class variables question

Then:

```
Mutable A, B;
```

```
A = new Mutable("something");  
B = A;  
A.String = "something else";
```

In `_that_ case`, then yes...`B.String` will now also return "something else".

```
But if you do  
string A = "something";  
object B;  
B = A;  
A = "Something Else"
```

A and B point to different locations whereas A will be "Something Else" and B will be "something". Both A and B are pointers in both examples.

Assuming your first example did what you wanted, then the second example still would not. That is, simply changing the type of B from "string" to "object" doesn't cause boxing to happen. Applying the same idea to the valid example I offered, you get something like this:

```
Mutable A;  
object B;  
  
A = new Mutable("something");  
B = A;  
A.String = "something else";
```

In that case, your statement that "B will be 'something'" is not true. B continues to reference the same instance as A, and having changed that instance so that it contains "something else" instead, the instance of Mutable that B refers to will contain "something else" (because it's the same instance).

I got it.

If the string is inside the object and A and B both pointed at the object even if the string changed INSIDE the object, since A and B are both pointing at the same object with the string changed and therefore `A.String` and `B.String` would be the same.

But in the above case:

Re: Abstract class variables question

```
string A = "something";  
object B;  
B = A;  
A = "Something Else"
```

You really do have 2 references to 2 different objects (a string and an object) and NOT 2 references to the same object. And because of the way a string works when you change A you actually get a 3rd object and the original object is dereferenced by 1 and if nothing else is referencing it, the GC will get rid of it.

Now, as long as you keep that reference in a variable typed as "object", you have no easy way to get at the contained string. But it's there, nonetheless. And you can in fact get it back simply by casting B back to the Mutable type that it is:

```
Mutable A, C;  
object B;  
  
A = new Mutable("something");  
B = A;  
A.String = "something else";  
C = (Mutable)B;  
Console.WriteLine(C.String);
```

This will print "something else" to the console. More significantly, once you've initialized A, the actual order of the change to A is irrelevant. The only requirement is that A is initialized before being assigned to B, and B is initialized (assigned) before being assigned to C. The line that changes the instance can be anywhere after A is initialized, and the code will have the same exact output, and in fact will do all of the exact same work (just in a different order).

Confused yet? :)

I think I have it. In the above example, I think $A = B = C$, since all 3 are reference types and point to the same object (and none are strings).

Here's the thing: what you've posted seems to imply that you believe that any time you assign something to "object", it's boxed. But that's not true at all. Boxing *only* happens when you assign a *value* type to an "object" variable. Because of the inheritance that is allowed with reference types, and because "object" is the lowest base class for every reference types, any reference instance can be assigned to a variable of type "object" without any conversion at all. It's already an "object".

For reference types, it's no different than if you assigned a "BoolType" instance to a "DataType" instance (using the classes we started with

Re: Abstract class variables question

here). No conversion happens, the reference is simply assigned from the "BoolType" variable to the "DataType" variable.

The "object" type is also the base class for value types, but only in a strange, "kind of" sort of way. Value types don't have inheritance, so it's sort of weird to say that a value type has a base class. But in C#, they do. Even so, because value types aren't references, you can't assign a value type to a reference variable typed as "object" without doing something to bridge the gap.

Boxing is what does that bridging. It copies the value into a whole new data structure, a structure that is itself a valid reference type. Only `_then_` can the reference, newly created, be assigned to a variable of type "object".

Now, back to the example you tried to construct, for it to work you first need to start with an actual value type, so that some boxing will take place. If you do, you might wind up with something like this:

```
int A, B
```

```
A = 5;  
B = A;  
A = 10;
```

Of course, for the example to work you'd like for changing A to also change B. But that's not how value types work. So, even though the above example hasn't even had the boxing introduced to it, you've already got a situation where changing the original doesn't change the previously assigned variable.

In that respect, other than the overhead of the boxing, boxing is actually very much like how value types operate normally. Yes, boxing creates a copy of the value type. But then, so does any assignment of a value type. And likewise, since a boxed value type is immutable, even after you've boxed a value type, there's no way to have multiple variables reference the data in a way where you can change the data via one variable and see the change via a different variable.

In other words, it's only going to be confusing if you think boxing might do something above and beyond just creating a reference-able instance of a value type.

Maybe it is different with ints and strings.

The int type and string type are `_definitely_` handled differently. An int is a value type, and so to assign an int instance to an object variable, it has to be boxed. But a string is a reference type, and assigning a string instance to an object variable requires no such work.

Re: Abstract class variables question

But the string is a special type of reference type (immutable) in that you can't just assign the reference to another string and assume they will stay the same, since any change to the string will now change the object the reference is pointing to. Correct?

Thanks,

Tom

Pete