

Re: Casting a parent class to a child class

Source:

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.csharp/2006-10/msg02267.html>

- *From:* "S. Lorétan" <[tynril@\[nospam\]gmail.com](mailto:tynril@[nospam]gmail.com)>
 - *Date:* Thu, 12 Oct 2006 08:51:48 +0200
-

Very, very good explanation. I think I've now understood what a cast is really, and the fact that the datas of a class instance aren't changed, but only the view of these datas. Thank you very much, I feel better now that one of my autodidact's error seems corrected!

Bests regards, and thank you again.

"Peter Duniho" <NpOeStPeAdM@xxxxxxxxxxxxxxxxxxxx> wrote ni:
12iqe42bstInhcd@xxxxxxxxxxxxxxxxxxxx

"S. Lorétan" <[tynril@\[nospam\]gmail.com](mailto:tynril@[nospam]gmail.com)> wrote in message
news:%23MkObqR7GHA.3384@xxxxxxxxxxxxxxxxxxxx

Ok. But I don't understand why it is designed this way.

It seems that may be simply because you're looking at class inheritance the "wrong" way (that is, differently from how it's viewed in the general paradigm of object-oriented programming).

For me, there is data loss during the cast from a Cat to an Animal.

This is what I mean. :) You don't lose data when you cast an instance from a Cat to an Animal. All you do is change your **view** of the data. All of the original data is still there, but the reference to the data that is of the type Animal only sees the parts of the data that are common to all Animals.

When you cast it back to the original type, your view changes and you get to see all of the data that was originally there, and was always there even after the instance got cast to an Animal.

I suspect part of the confusion lies in the fact that C# includes some very robust type-conversion functionality that shares the casting syntax. It's important to keep in mind the difference, however. When you "cast" an int to a string (for example), this isn't really a cast...it's a type

Re: Casting a parent class to a child class

conversion that just looks like a cast. This might lead one to believe that that's what casting is all about: converting one type to another. But in reality, true casting doesn't change the data at all. It just changes how you treat the data.

But there isn't during the (impossible) cast from an Animal to a Cat, because Cat inherits all the attributes of Animal *plus* some cat-specific ones.

The Cat *class* inherits all of the attributes and behaviors of the Animal class. But it has its own attributes and behaviors as well (and may modify the generic attributes and behaviors of the Animal class). These attributes and behaviors are decided when the *instance* of the class is created. Casting cannot change this.

The problem with casting something that is an Animal to something that is a Cat comes when that something wasn't a Cat in the first place. You can't treat just any old Animal as a Cat. You can only treat an Animal that *is* a Cat as a Cat.

So, in the example that was given:

* You can treat a Cat as an Animal, because it is. When you do this, you ignore the things that make the Cat a Cat, paying attention only to those things that Cats share with all other Animals (including Dogs).

* If you have a "something" that you're treating as an Animal, but which you know is actually a Cat, you can cast that something to a Cat. In doing so, what you're saying is "I've got an Animal, I know the Animal is actually a Cat, and now I'm going to start paying attention to the things that make it a Cat".

You appear to be feeling as though when you cast something from an Animal to a child class (such as a Dog), that more-derived reference should be somehow populated with the Animal parts, along with new default Dog parts. But that's not how it works. When you cast something, you're casting an *instance*, not the class itself. The instance has already been determined as to what type it actually is. The only thing that casting does is change what your *view* of that instance is. It can't change the data that's actually in the instance.

So, this means that in the example given, the *instance* created was a Cat. You can treat the Cat simply as an Animal for a time. After all, all Cats are Animals. Like all Animals, they need to eat, they need to sleep, they have characteristics such as mass and gender. These are things common to all Animals, and if you look at only those characteristics, you don't need to know that the instance is a Cat. Knowing it's an Animal is sufficient.

Re: Casting a parent class to a child class

But the instance is still a Cat, and if you start treating it like a Dog, you get into trouble. A Cat won't chew on a bone, it won't fetch your newspaper, and it won't bark. So your instance, currently being viewed as an Animal but actually created as a Cat, cannot do any of these things. Casting it to a Dog doesn't change the fact that it's actually a Cat.

I feel much more natural with this way. I understand that there can't be both casting (Child to Superclass and Superclass to Child), but the Child->Superclass cast as C# implements feels strange to my logic.

It's not just that C# implements it this way. All object-oriented languages handle it this way. As far as "there can't be both casting" goes...I'm not sure what you mean by that, but in OOP the casting *is* bidirectional. In the example given, you can cast something that starts out as a Cat to an Animal, and then cast it back to a Cat. Then back to an Animal. Then back to a Cat. Over and over, as many times as you like.

At no point during all that casting does any of the actual data change. All that changes is how the program views the data. Sometimes the code only cares that the instance is an Animal, and in those cases it will use the instance cast as an Animal. Other times, the code is dealing with Cat-specific things and in those cases it will use the instance cast as a Cat.

Let me explain myself.

```
Animal test = new Cat(); //Valid because a Cat IS an
Animal
Dog test2 = (Dog)test; //Whoa, can't change a Cat into
a Dog!!
```

I don't think it this way. On the line 1, we cast a Cat to an Animal. So, the Cat object *lose* its specific attributes.

But it doesn't lose its specific attributes. It retains everything about itself that was Cat-like. The only thing that is "lost" is that the code cannot access the Cat-like attributes while using the Animal-typed reference.

Then, on the line 2, we cast an Animal (with *only* the Animal's attributes) to a Dog, which is valid, because the Dog class implements every attributes of the Animal class because it's a child class.

Re: Casting a parent class to a child class

But the **instance** isn't a Dog. It's a Cat. When you try to cast the instance, originally created as a Cat, to be viewed as a Dog you are making an error.

Your impression of what a cast does is fundamentally flawed. Casting something doesn't change what that is. All it does is change how you perceive that thing. You can try to perceive a Cat as a Dog all day long, but it will still be a Cat.

Here is the way I'm thinking about:

```
Animal an = new Animal(); // We create an Animal with
basics attributes
setting to their default value in the constructor...
Dog dog = (Dog)an; // And we add some Dog-specific
attributes to this
object while casting it to a Dog.
```

In this way, there is no data-loss. We can't cast a Dog to an Animal, because not Animals are Dogs. But we can cast an Animal to a Dog, because every Dog are Animals!

But you can't do what you're describing. In order to cast an object instance to a specific type, it needs to already **be** that specific type. In your example, you've created an instance of a generic Animal. This instance will never be able to be a Cat **or** a Dog, because it has none of the characteristics of either.

Casting doesn't "add some <class>-specific attributes" to an object. The object needs to already have those attributes for the cast to be successful.

I don't know if I am understandable, I'm not very comfortable with the english language.

IMHO, your English is fine. I do believe that you've explained your view of casting correctly. I'm hopeful that the above helps you understand why that view is incorrect, and to see what the correct view is.

And for what it's worth, the OOP way of thinking is well-established, but that doesn't mean it's necessarily intuitive. For someone trying to learn how OOP works, it's perfectly understandable that there may be some trouble getting a good grasp on some of the basic concepts. Just keep in mind that it's your job as a programmer to come to terms with the way the language works, rather than to invest too much effort believing that the

Re: Casting a parent class to a child class

way the language works is wrong (unless what you want to do is design a computer language :)).

Even if the language behaves in a way that isn't intuitive to you, at end of the day you still need to comply with the language's requirements in order to use the language. That will go more smoothly if you can get yourself into the same frame of mind that the language designers intended, rather than holding tightly to your own preconceived notion of how things should be. :)

Pete