Please preserve cute trees! Don't print this document unless necessary!

FluentOMapper (FOM) – Reference V1.0

Presentation

Basically**, FluentOMapper** (FOM) is a framework to describe an execute mappings from objects to others using their properties:

- **Fluent**: Using chained method calls to describe the mapping.
- **O** for Object and like old "O'Matic" expression.
- **Mapper**: Because you map objects to others

FOM is strongly typed and uses lambda expression to describe mapping, so you can use refactoring of visual (or other tool).

A typical use is for writing DTOs, but not only ;-)

History

I've been working in Java many years, and now as an expert. I've met many interesting concepts, like DTO mappers.

Now, I'm working in dotnet for a few years, and I've met the same needs than in Java: Mapping objects to others.

The power of C# (lambda expressions, good management of generics), more than Java (no troll, I love Java and J2EE), opens some possibilities that I've tried to share via FOM.

Other "kind of" API exist.

While writing this API, I tried to make "simple things to do ➔ simple to describe", by applying the nice sentence: Convention over configuration.

Bus the more I studied cases, the more complex situations appear. So I implemented advanced capabilities with tuning methods, keeping in mind that using FOM must not be more complicated that making your own bunch of code (in most cases I mean).

My ultimate goal was using the FOM tools to clone an object tree, whatever this can be. (Ex: cloning read only collection or interfaces). I think that it's now possible (see cases in this doc).

So my work has led to a kind of extendable conversion framework, which the most basis case is the object property mapping! But some pieces of brain may be consumed understanding tricky cases.

But with FOM, because a same need follows the same patterns, it's useless to think twice a pb, this will avoid tons of coding to maintain that would lead to some mistakes, by delegating it to a (reliable? I hope!) framework.

Features

- Map an objet tree to another: taking each described properties from a source and set it into an existing target. If not target specified, FOM will create a new set when required by the mapping. This distinction is very important because FOM may behave differently according to an existing target or not.
- Create new instance of a mapped property instead of using the existing one. Used to do partial or total replacement.
- Override source and/or target type. Useful when mapping from/to interfaces or abstract, or to convert from type to another.
- Map arrays/list … to another (using embedded converter), and map inner objects of the array to ensure continuity.
- Use embedded converters, or create yours, to achieve complex / specific mappings.
- Use condition (or extend yours) to apply mapping or not.
- Cache your mapped objects for advanced capabilities
- Tune some behaviors to match your needs.

What does it not cover?

The first thing to say: FOM can't cover all the needs of the entire world. Basically, FOM was design to copy a property to another, with some tuning options.

Time running, FOM expanded to support more features you can meet in you developments (array, over-typing, etc …).

Specific models will require a specific coding. Even if you can model your need with FOM, it would be prettier to do it yourself instead of twisting your brain with FOM.

For example, in a massive DTO environment, FOM helps you saving time.

I like to imagine that 90% of the cases can be solved by FOM. Let me know!

Reference FOM in your project

Simply add a reference to **FluentOMapper.dll**. Nothing else required. No log API.

Usings

Basic usings (basic programming)

```
using FOM.Impl;
using FOM.Interfaces;
```

If you plan to use embedded converters (advanced programming):

```
using FOM.Impl.Converters
```

For catching FOM Exceptions:

```
using FOM.Impl.Exceptions
```

And to use some utility classes (expert programming):

```
using FOM.Utils
```

Instantiate the manager

The manager is the "start" object, used to register mappings and get a hook on each one.

Standard instantiation

```
IManager mngr = new ManagerImpl();
```

Using Unity (IoC)

```
_UnityContainer.RegisterType<IManager, ManagerImpl>(new ContainerControlledLifetimeManager());
```

Register "myFirstMapping"

Mappings are registered by name in FOM. To create one:

```
IMappingNode<ObjectA, ObjectB> rootmap = mngr.RegisterMapping<ObjectA,ObjectB>("myFirstMapping");
```

➔ This tells FOM that you have a mapping named `myFirstMapping` that will be from an object `ObjectA` to object `ObjectB`.

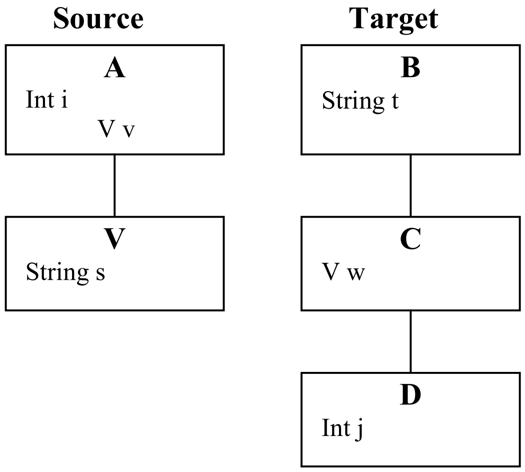
➔ `rootmap:` You get the root node of your mapping. From this one, you can go on describing your properties and child-object mappings.

Starting the mapping description

Surprise! Via the previous mapping registration, you already have created a mapping. That's what I call **object mapping** (object to object). The other mapping type (and mainly used) is the **simple property mapping**:

Simple property mapping

Consider the following object model:



And you need:
- To copy prop **`A.i`** to prop **`D.j`**
- To copy prop **`V.s`** to prop **`B.t`**
- To copy prop **`A.v`** to prop **`C.w`**

So, you will write

```
rootmap.Map(a => a.i, b => b.w.j);
rootmap.Map(a => a.v.s, b => b.t);
rootmap.Map(a => a.v, b => b.c.w);
```

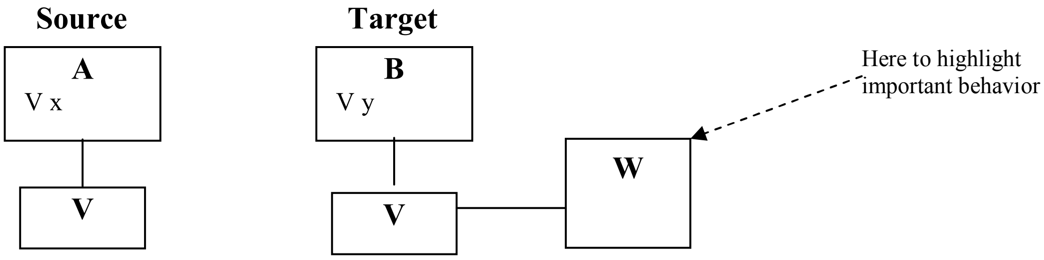
➔ That's all :-). The default behavior is copying reference for object, and value for value types.

The two concepts of mappings:

I previously said that two kinds of mappings exist:

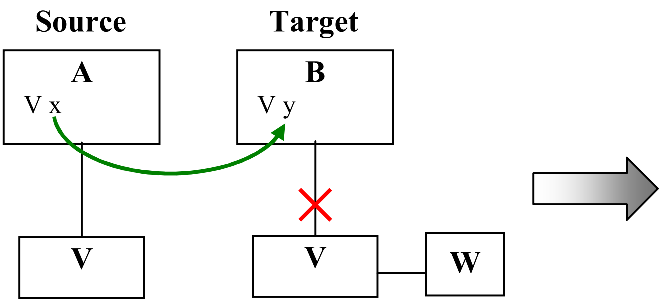
- **Simple Property mapping**: `Map(src➔src.x, dst➔dst.y)`. The widely used. This means that property **x** in **src** object will replace property **y** in **dst** object:
    - If the prop is value type or string ➔ value is copied
    - If the prop is Object ➔ reference is replaced.

Example:



Here to highlight important behavior

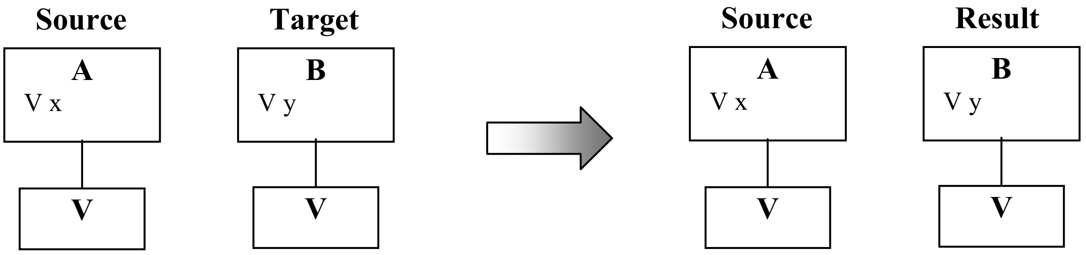
`Map(a=>a.x, b=>b.y)` does the following:



1) New source reference of V in A replaces reference in B object

<u>2) Important</u>: The **W object** has disappeared. This is logic because it was attached to the previous instance of V in B. => FOM does not relink child objects (by default ;-).
=> *<u>Advanced</u>: This can be overridden by using our embedded converter called **ReconnectOriginalChildrenConv***

- **Object mapping**: `Map(src=>src, dst=>dst`. This is used in special cases, like root mapping (always existing when creating your mapping), some array mappings, conversions and GoTo instruction. We will see concrete use cases later.
  But let's imagine It's like a new "root" start point, so that the existing target object (if given to the mapping engine) is kept as it (and can be used for next mapping), instead of being replaced by the source like a standard mapping pointing at the object:



```
       mngr.RegisterMapping<ObjectA, ObjectA>("o2o") // declare it
             .GoTo(a => a.x, a2 => a2.y) // move cursor to target type V
             .Map(s => s, d => d); // map V on target as "itself"
```

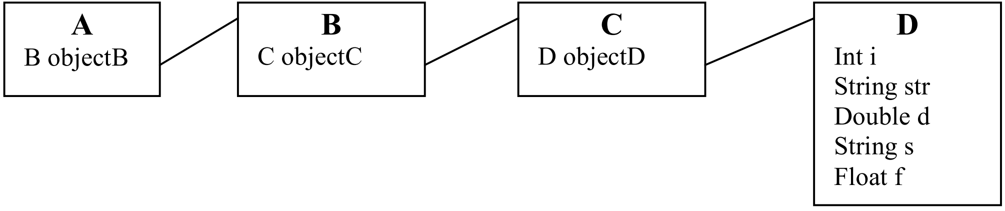
➔ Like the previous example on property mapping, It's "x ➔ y" which is globally addressed. But the **GoTo** just seek the object navigation directly to x and y (no property mapping, just a seek), and the **Map** says that a new kind of "root" start point for mapping is defined.
It seems useless, but don't forget: it's a mapping anyway. So from this point, you can apply the altering methods like newInstance, OverridingType, and moreover UseConverter.

Can I map multiple times the same path?

Yes.
In this case, the declaration order is used. This allows subdividing treatments, but would not be used every time.

GoTo: Navigate into the model paths

GotTo is used to navigate to a source and target object, and then, to go on mapping from this new point.
Let's consider the following object set:



You want to map the D properties (on another D object), root from A. You can write:

```
rootmap.Map(s=>s.objectB.objectC.objectD.i, d=>d.objectB.objectC.objectD.i);
rootmap.Map(s=>s.objectB.objectC.objectD.str, d=>d.objectB.objectC.objectD.str);
rootmap.Map(s=>s.objectB.objectC.objectD.d, d=>d.objectB.objectC.objectD.d);
etc …
```

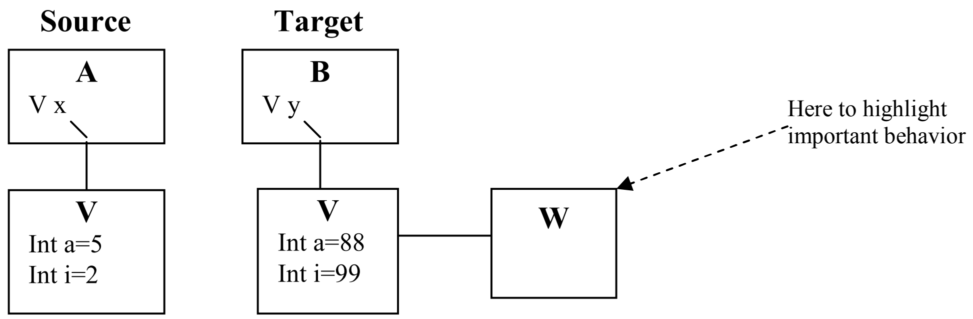
Or use **GoTo** to point at D object before mapping:

```
var tmp=rootmap.GoTo(s=>s.objectB.objectC.objectD, d=>d.objectB.objectC.objectD);
tmp.map(s=>s.i, s=>s.i);
tmp.map(s=>s.str, s=>s.str);
tmp.map(s=>s.d, s=>s.d);
etc …
```

As previously said, GoTo do not create mapping, so **`NewInstance()`**, **`UseConverter()`**, **`MapFromCache()`** and **`UseCondition`**`()` can't be called after a GoTo. But other functionality can be called (Map, Exclude, GoTo, Override source and target types).

Create a new instance

You can create new instance for the target instead of taking the existing one in the source.
Note: This behavior is valid either from a property mapping or an object mapping.
Let's imagine the following:

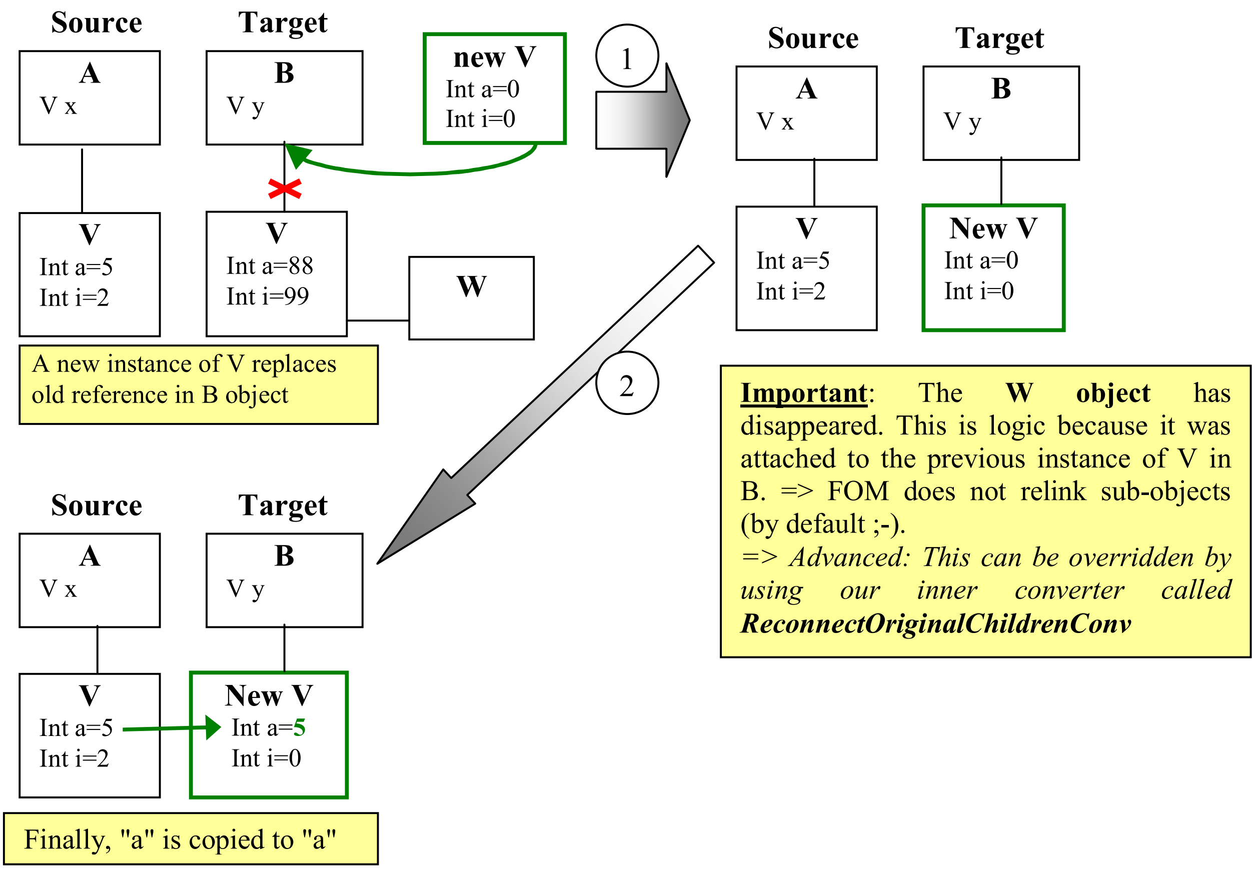


Here to highlight important behavior

You want to Map A ➔ B, and A.v to B.v by creating new instance of V, and finally, map V.a to V.a (only, not "i")

So you write:

```
mngr.RegisterMapping<ObjectA, ObjectA>("MyNewInstanceMapping") // declare it
      .Map(a=>a.x, b=>b.y).NewInstance() // y will be replaced by new instance
      .Map(vsrc=>vsrc.a, vdst=>vdst.a); // copy "a" value
```



A new instance of V replaces old reference in B object

Important: The **W object** has disappeared. This is logic because it was attached to the previous instance of V in B. => FOM does not relink sub-objects (by default ;-).
=> *Advanced: This can be overridden by using our inner converter called **ReconnectOriginalChildrenConv***

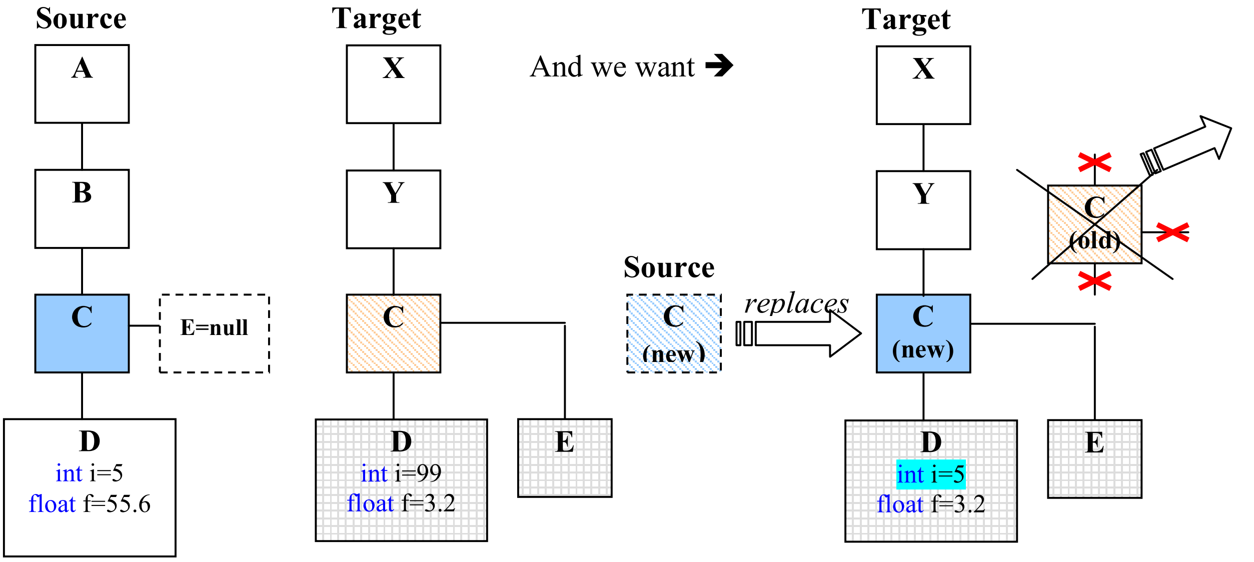
Finally, "a" is copied to "a"

Reconnect children of original object

As seen in previous examples (with or without *NewInstance*()), when replacing a reference in the target object, all children of the original (i.e. replaced) object are lost. This is logic, that's the principle of C# or even Java reference notion.

But using the **IConverter** mechanism is a way to replace just a part of the object set. The embedded converter that can do that is **ReconnectOriginalChildrenConv**.

NOTE: IT ONLY WORKS WITH NEWINSTANCE(). OTHERWISE, SOURCE OBJECT WOULD BE ALTERED!!! LOOK AT THE EXAMPLE TO UNDERSTAND.

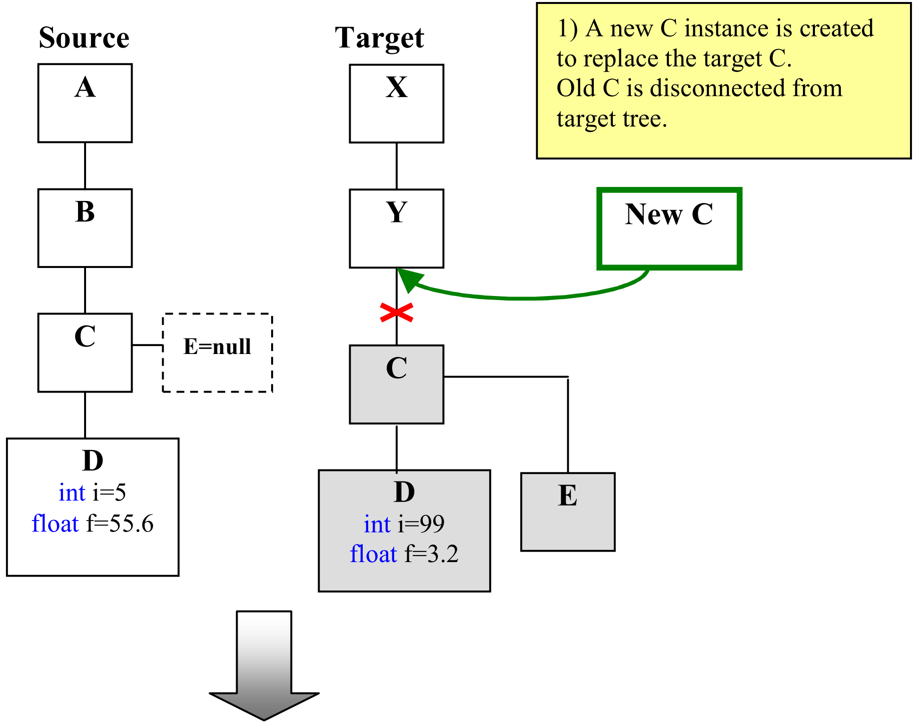


We want to replace C object of the target source into the one in target, but we want to maintain the original target children objects "D" and "E" instead of source C children.

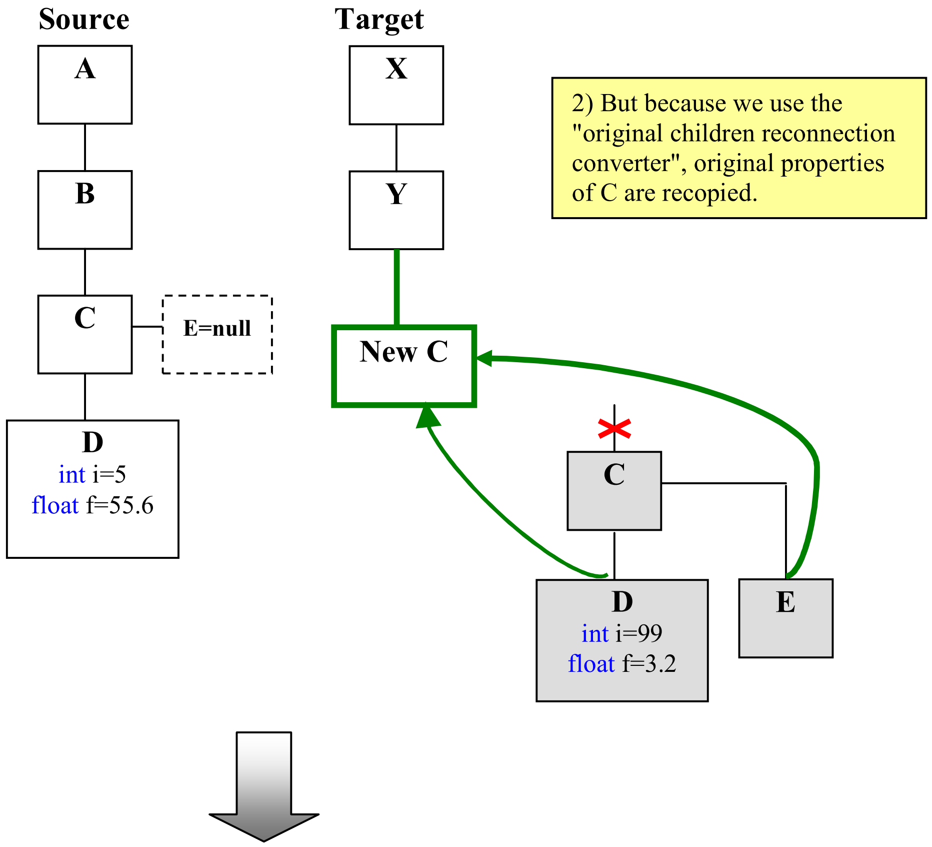
Bonus, for the fun, we want to copy the *int value* in source D (=5) to the target D.

```
rootmap.Map(source=>source.a.b.c, target=>target.x.y.c) // adress c prop
      .NewInstance() // Mandatory when using ReconnectOriginalChildrenConv
      .UseConverter(new ReconnectOriginalChildrenConv()); // it's here!
      .Map(ds=>ds.d.i); // Bonus: copy the source D value for property "i"
```

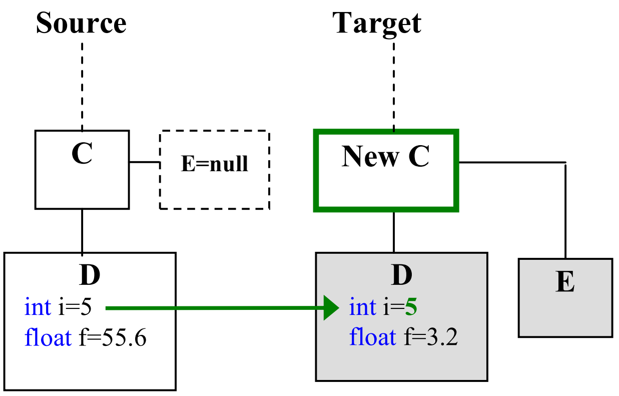
This will produce the following behavior:



1) A new C instance is created to replace the target C.
Old C is disconnected from target tree.



2) But because we use the "original children reconnection converter", original properties of C are recopied.



3) Finally, the "i" value is got from source and put to the D target.

➔ Now, you understand why only **newInstance()** is eligible for **ReconnectOriginalChildrenConv**. The original source object would have been altered with children target object. And the concept of FOM is to map from source to target, not the inverse ;-)

➔ It looks like something very tricky, but, I use to say: "You will understand this case as soon as you meet the need".

Type overriding

You use the type overriding in two cases:

- When the property (source or target) is an interface
- When the target property is not the same as the source, and (generally), you apply a converter.

To do that:

```
map.OverrideSourceType<MyobjImpl1>(); // overriding source type
map.OverrideTargetType<MyobjImpl2>(); // overriding target type
```

Overriding type is useful because fluent continuity is maintained (i.e. children mapping from overridden type goes on using overriding type, and not the original type).

FOM is not magic: To make an overridden type to be applied, there must be a way to do so:

- By implicit sub typing (of interface, inheritance, etc …)
- By a converter that knows how to deal with the types.

Execute the mapping

Short solution: Just call:

```
var res=(ObjectB)mngr.ApplyMapping("myFirstMapping ", srcA, srcDest);
```

➔ srcDest is the receiving object (for example a DTO)

But if you call:

```
var res=(ObjectB)mngr.ApplyMapping("myFirstMapping ", srcA); // no target!
```

➔ FOM will create the resulting tree for you. Nice?

So the question is: Why do not use the second syntax only, it's automatic? Because you can merge objects. Imagine **`srcDest`** already contains some fields, and you just want to copy some others ➔ Use the first syntax (It's what I call a surgical strike!).

Path Exclusions

The question: Why excluding paths? **I, me, personally,** decide what to map!

That's right! If you set a mapping and then exclude it, the mapping is ignored (exclusion takes precedence to mapping).

➔ Told like this, you think: "Path Exclusion is just useless!"

In fact, exclusion paths are mainly used for Converters. For example, FOM embeds a converter (see below) called **PropertyCopierConv** that copies all object properties to another object.

Combined with exclusion path, you can say *"I want to copy all properties, except…"*

So imagine an object "**A**" with 10 properties **a,b,c,d,e,f,g,h,i,j**, and you want to map all of those on **another A** object, excepted the "**g**" property.

You can write:

```
var rootmap = mngr.RegisterMapping<ObjectA>("NoGMapping");
rootmap.map(s=>s.a);
rootmap.map(s=>s.b);
rootmap.map(s=>s.c);
rootmap.map(s=>s.d);
rootmap.map(s=>s.e);
rootmap.map(s=>s.f);
// no "g" prop
rootmap.map(s=>s.h);
rootmap.map(s=>s.i);
```

Or you can write:

```
var rootmap = mngr.RegisterMapping<ObjectA>("NoGMapping")
.UseConverter(new PropertyCopierConv ())
.Exclude(s=>s.g);
```

➔ The converter will map all properties, AND will ignore the excluded paths. This works this way because the converter knows how to deal with the exclusion paths. This is not magic.

Converters

Embedded converters

I have thought about some converters that I found useful. So, they are embedded in the library. But you can create yours (see below).

- **StringToValueTypesConv**: It can take a string as source, and convert it on value type (*int, float, double, long, boolean, enum and string*). Useful for string only DTOs.
- **ValueToStringConv**: Take any source object, and returns the ToString() call. Useful for string only DTO.
- **PropertyCopierConv**: This converter takes properties of the source object, and set it to target object. Property matching is done by name.
  - If the name does not exist in the target, it's ignored.
  - This converter is useful for auto-mapping. It takes path exclusion in account.
- **ListCrossConv**: Convenient! Use this converter to
  - Convert from object "IEnumerable" to another object "List". By List, I mean HashSet, IDictionnary, IList or fixed Array.
  - To submap objects in the list and ensure continuity in the mapping objects set. Sub-mappings are provided to converter at the `new`. If the object of the list is an instance of a source sub-mapping type, the mapping is called on the object. Priority is: First match, first taken.
  - Supports object caching.
- **ReconnectOriginalChildrenConv**: This converter does not apply from source to target object, but from original object in target to the new object that will be set in the target. It copies each child (references and value types) of the original object to the replacement object. Matching is done on property names. It takes path exclusion in account. To avoid source alteration, this is only possible on new instance created for target (NewInstance() call).

➔Note that you can enrich your mapping, even after a converter.

Extends with your own converters

IMPORTANT: Because IConverter interface give you the hand to all the objects, you can write destructive code on your objects or put the mess in the engine. That's said.

You can imagine complex converters, to match special cases. To do this, you need to implement the IConverter interface, which have one method:

```
object Convert(MappingContext ctx);
```

The mapping context is given by the engine, and this object contains:

`Type` **`Tsrc`**: the expected source type. That's not the type of the real source object, but the Type declared (or overridden) in the mapping description.

`Type` **`Tdst:`** the expected destination type. That's not the type of the real target object, but the type declared (or overridden) in the mapping description.

`object` **`ObjectSrc`**: the source object to convert from.

`object` **`ObjectDst`**: the destination object to convert to.

`object` **`OriginalObjectDst`**: This is the object that will be replaced in the target. So it can be null.

`IPathLink` **`PathLink:`** The source and target path (as string dot separated, and array). These paths are absolute paths (i.e. from the root object).

`List<IPathLink>` **`ExcludedNodes:`** The node declared as excluded from the mapping via the Exclude instruction. It can be null if no node defined.

`IConverter` **`Converter`**: The converter used (if any)

`ICondition` **`Condition`**: The condition on this node (if any)

`object` **`TopSourceObject`**: The source object given to the method **`ApplyMapping`**.

`object` **`ToptargetObject`**: The target object (if any) given to the method **`ApplyMapping`**.

➔ In return of the method, you can have … well … whatever! But in most of the cases, returning the source object itself, after conversion operations, is a good start point.

➔ Look at the existing converters code to have an idea on how to begin.

Arrays/List/Dictionnary/Hashset

FOM allows powerful processing for array. It can:

- convert from one type of array to another (using embedded converter mechanism)
- Start a sub mapping on inner object of the array

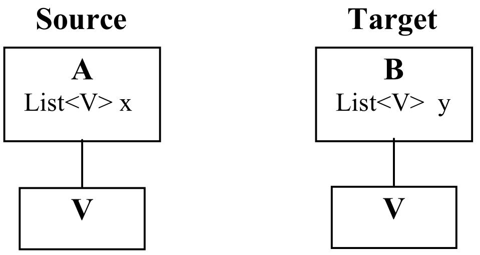
For this, FOM provides a converter called **ListCrossConv**.

Map on Array/List/Dictionnary: Copy reference or content?

Rapid answer is: Reference copy of the array, like any other object. So, why a special chapter? Because here are meaning considerations of the world "mapping" for array:

Reference Copy

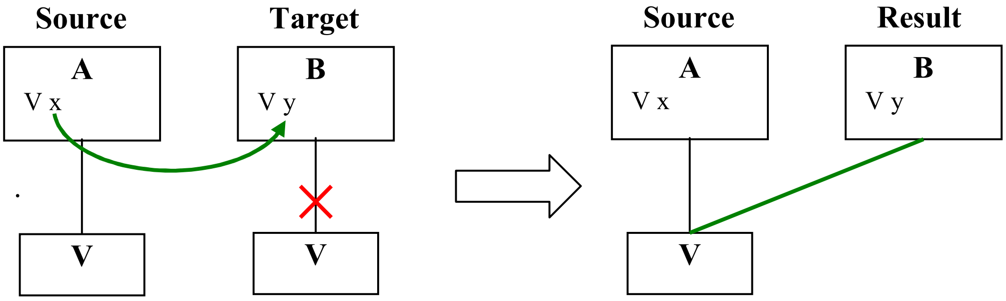
Imagine the following object tree:



We want to **"copy"** field "x" to "y". Naturally, we think to write:

```
Map(A ➔ A.x, B ➔ B.y)
```

But x and y are both objects (In our example, a **List<V>**). But in FOM definition, mapping this way (by field) means that reference is copied. So the result of such a mapping is like any other basic field mapping:



➔ A and B holds the same List<V> reference. This is the default behavior.

Content copy

But when writing the mapping, you probably mean: "I want to copy the array content to the other existing (or new) array". But how FOM can do this?

The best way is to use the "**GoTo"** to target the List Object, and use the object mapping:

```
GoTo(A ➔ A.x, B ➔ B.y) // seek to x and y list props
      .Map(s➔s, d➔d) // map object to object to be able to use a converter
      .UseConverter(new ListConverter())
```

This tells FOM to:

- **GoTo**: Point to the source and destination object directly. It does not Map the property (And that's what we want here), but set the dept cursor to the properties.
- **Map**: We map an object on one another. So, this is not a property reference copy, but an object to another object mapping. (Like a root object mapping f.ex).
- **UseConverter**: To use our embedded converter that can convert any array into another.

This leads to multiple possibilities:

- If the destination array already exists in the target object tree, it will use it. So, objects will be added to existing list. This is a kind of Merge. If you want to override this, use the **NewInstance().**
- If not, it will be created according to the declared target array type. If the target array type is an interface, the TargetType should be provided the following ways:

```
mngr.RegisterMapping<A, B>("testArray1")
      .GoTo(A ➔ A.x, B ➔ B.y) // seek to x and y
      .OverrideTargetType<List<V>>() // specify the type for new object
      .Map(s➔s, d➔d) // map object to object to be able to use a converter
      .UseConverter(new ListConverter());

      // Or //

mngr.RegisterMapping<A, B>("testArray2")
      .GoTo(A ➔ A.x, B ➔ B.y) // seek to x and y
      .Map(s➔s, d➔d) // map object to object to be able to use a converter
      .OverrideTargetType<List<V>>() // specify the type for new object
      .UseConverter(new ListConverter());

      // Both have the same meaning //
```

If not mentioned, FOM will use the source instantiated type.

Conditional mapping

How to use a condition?

Shortly, apply method `UseCondition(IConverter converter)` on a mapping node.
If the condition is true, the mapping is realized according to the they various options.
If the condition is false, the target object is left "as is" (i.e. if not target object provided ➔ null).

The condition is tested at the end of the other option (NewInstance, UseConverter, etc …). So even if the condition is false, some processing may occur.

Does condition applies to children?

Choose it!

In the **ICondition**, the property **IgnoreChildMappings** can be set.

- If false, the child-path are processed, even if parent path were false
- If true, all the child-paths starting from the conditioned path are not processed if conditioned path did not apply.

Example

```
root.Map(a=>a.b).UseCondition(xxx); // with condition xxx returning "false"
root.Map(a=>a.b.c).Map(x=>x.z);
root.Map(a=>a.b.f);
root.Map(a=>a.c);
```

If **IgnoreChildMappings** is false ➔ `Map(a=>a.b)` will not be executed, but other are.
➔ The resulting behavior is like `Map(a=>a.b)` has never been declared.

If **IgnoreChildMappings** is true ➔ `Map(a=>a.b)` will not be executed, and other under *a.b* are not, but *a.c* is executed (because not under a.b)
➔ The resulting behavior is like `Map(a=>a.c)` has been the only mapping.

Q: Is it possible to force a specific childpath to be executed even if parent is not?
➔ *No: Use another new mapping over the previous. There is a time we must stop …*

Embedded conditions

`IsNumberEqualsCond``(double testNum)`: Returns true if the field is a source field is a number AND equals to a specified **testNum**. Child mapping are ignored if true (supposed to be value type, so ...)
`IsObjectCond`: Returns true if the source field is an object. If false, child mapping are ignored, because if it's not an object, how could child mapping can exist?
`NotInCacheCond`: Returns true if the source object is not in the cache. It's useful to "break" some duplicate behavior while maintaining references.

Extends with your own condition

IMPORTANT: Because ICondition interface give you the hand to all the objects, you can write destructive code on your objects or set the mess in the engine. That's said.

Just implement the following method:

```
object CanApplyMapping(MappingContext ctx);
```

And the following property:

```
bool IgnoreChildMappings { get; }
```

The mapping context is given by the engine, and is the same as the one provided for IConverter. Please see IConverter for detail.
If **`IgnoreChildMappings`** is true, children mapping are not applied.

Reusing already processed objects from cache

Description

If, in the source tree, two mapped properties with NewInstance (no other way) point at the same reference, you can tell the engine to remap on the same newly instantiated target property.

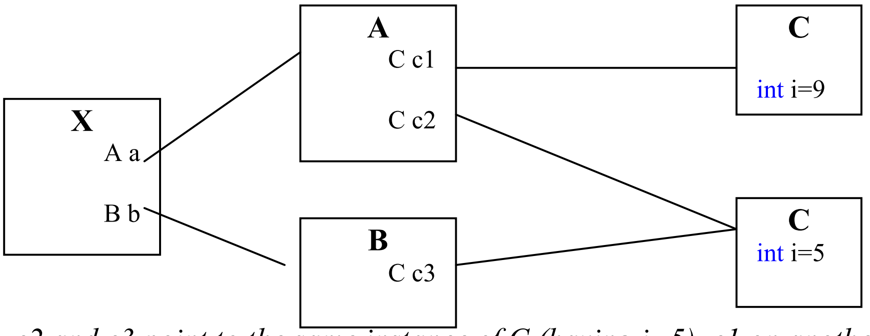
To use this possibility, use the NewInstance method like:

```
map.NewInstance(bool reuseCached)
```

`reuseCached` parameter tells the following behavior to the engine:

- If the source reference has already been processed (so a new instance has been created for this source reference somewhere in the object tree), it will be reused instead of creating a new one again. This allows relinking objects.
- If not already processed, new instance is created and associated with the source reference to become available to `next NewInstance(true)`.
- All the `NewInstance()` implied in the process have to be set to **`true`**. Otherwise, the newly created instance is not stored in cache and can't be reused later.

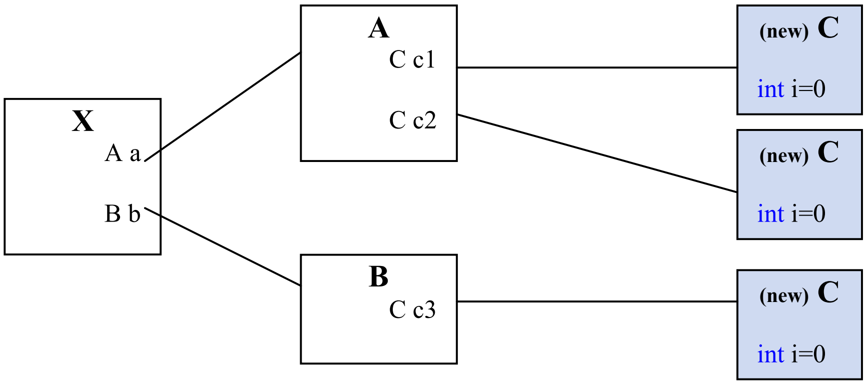
Example:
Imagine the following objects tree (we talk about "instances" as usual):



c2 and c3 point to the same instance of C (having i=5), c1 on another one (with i=9).
Now, basically, we want to create the following mapping and create new instances of C:

```
root.Map(x=>x.a.c1).NewInstance();
root.Map(x=>x.a.c2).NewInstance();
root.Map(x=>x.b.c3).NewInstance();
```

This will produce the following instance tree:

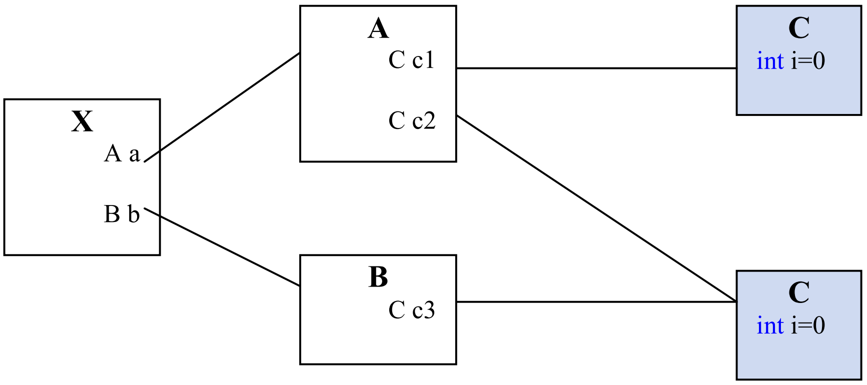


The first thing we can see is that the resulting tree does not look like original tree. How to tell the mapping to use the **same instance**?

➔ By providing "`true`" to `reuseCached` parameter of `NewInstance();`

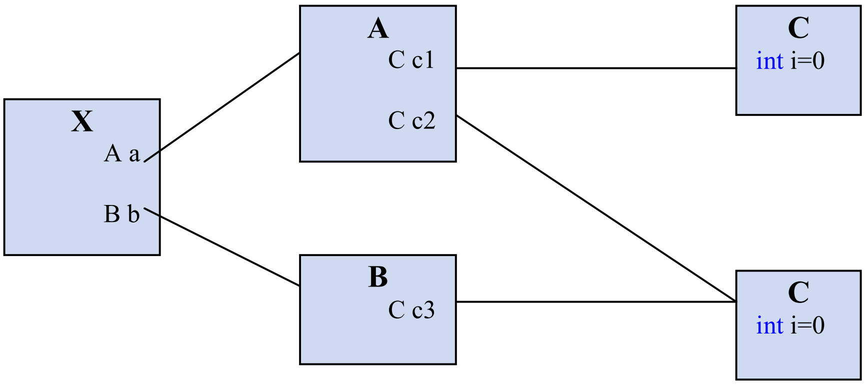
```
root.Map(x=>x.a.c1).NewInstance(true);
root.Map(x=>x.a.c2).NewInstance(true);
root.Map(x=>x.b.c3).NewInstance(true);
```

This will produce:



➔ That's the same as the original, except that C instances are freshly created.

➔ By extension, if you use NewInstance(true) on every object on every property, you get:



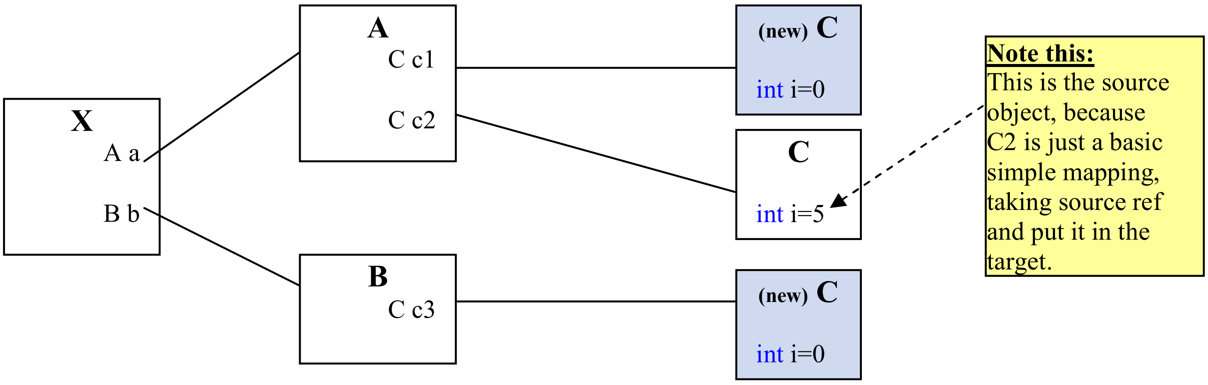
As we say in France: "*Cerise sur le gâteau*": You have deeply cloned you object tree, and preserved linked references.
Note*: It's not magic: It's because* ***ListCrossConv*** *do some job on cache. Refers to code for more information.*

What happens if I mix?

What did you expect? A non-newInstance will keep the source. So the mapping:

```
root.Map(x=>x.a.c1).NewInstance(true);
root.Map(x=>x.a.c2); //no new instance here
root.Map(x=>x.b.c3).NewInstance(true);
```

Will produce:


A
C c1
C c2
(new) C
int i=0
X
A a
B b
C
int i=5
B
C c3
(new) C
int i=0
Note this:
This is the source object, because C2 is just a basic simple mapping, taking source ref and put it in the target.

Is it possible to choose the instance to re-use?

No.
➔ Sometime, I just stop! Exotic behaviors require a specific converter (that gives you access to the mapping context), or a specific code. It's not possible to cover all the cases in a FOM.

➔ The only liberty is on the **true/false** parameter that may imply some interesting behavior.

Does it apply to converters?

Not automatically.

➔ Because converters are your own code, it's not automatic. Converters are given the mapping context, containing the cached objects.

So, the **ListCrossConv** supports this, because I coded it, and the object cache created by the mapper is propagated to sub-mappings.

*If you zoom on the **ListCrossConv** converter, "mapper" is called for each object of the list, and the top mapping object cache is given as an argument. And when the object cache is provided (a `IDictionary<object,object> in fact`), it will use this one provided instead of creating its own.*

Moreover, cache is not filled before converter (which is logical, see below). If converter is recursive (ex: the ***ListCrossConv***), you may miss the currently processed object, and process it again, leading to duplication of the same object. ***ListCrossConv*** has solved this problem by adding the current working node in the cache at the beginning of the conversion, and then remove it at the end, letting the engine cache do its job by storing the resulting object.

What about mixing cache reuse + condition and/or converters?

It works!

➔ Condition validation and converter are called after the cached object has been retrieve from cache. So, no problem!

Execution stack

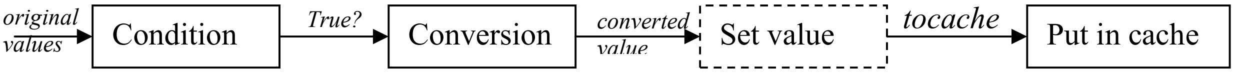
What's that?

As you may have understood reading the entire doc before, you can:

- Use a converter
- Use a condition to map or not
- Use the cache

But, a question if rising: In which order all these features are called?

By default, it follows this scheme (if a block is not used, imagine a bypass)



Condition is amongst all. If not true, the rest of the stack is not processed.
The converted value is the value put in cache.

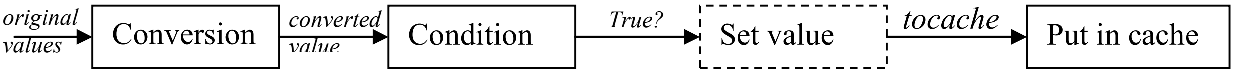
Can we override stack?

Yes. Condition and conversion can be inverted. Other can't, and it would be non sense. Setting the value is always done after conversion and condition, and caching always occurs when setting the value, (which is logical).

So, one method is available to inverse the conversion and condition order:

```
ConvertBeforeTest();
```

In this case, the conversion is done, and then resulting object is given to the conditional module.



As a summary, the difference is:

- **default case**: The condition is applied to the value before conversion, and conversion is not called if condition is not fulfilled. It's an interesting case when you want to avoid a time consuming converter.
- **Inverted case**: The conversion is called, and then, the condition is applied to the converted object. Original one is lost (sorry). So tests can be done on converted object.

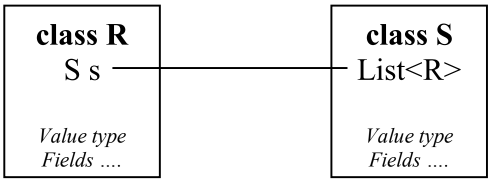
Can I put a condition before AND after?

No. If someone gives me 200$ or 180E, I can imagine myself working on it ;-)

Case studying 1: cloning 1-n relationship composition, attacking the mapping from the "n" side.

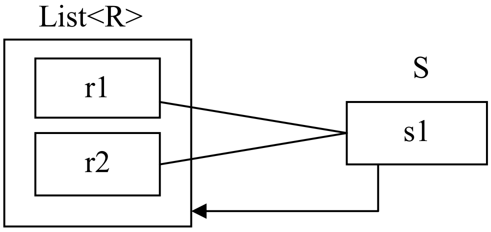
Presentation

We want to clone the following model:



Each R has a reference to S, S have a list of R.

And we consider the following tree instance:



It's a list of R {r1, r2}, each R pointing at the same parent unique instance of S "s". S instance "s" holds the list of R instances {r1, r2}.

➔ We want to clone starting from the List<R>
➔ Norma: White instances are from the source, gray are the new one created.
➔ Note: We can imagine r3 in another List<R> pointing at a s2. But we don't need to make the example more complex, but it would work the same way.

We will present the reflection path to achieve this, step by step. So this leads to a long example. But in fact, the resulting code in minimal and reliable, which is the aim of FOM ;-). This example is one of the most complicated for such a "simple" concept. But hard coding is worst.

Tip:
This example can be found in **TestUnits/UsingCacheTest.cs**
`Method:` **`TestCachePropagationToListConverterSIMPLE`**

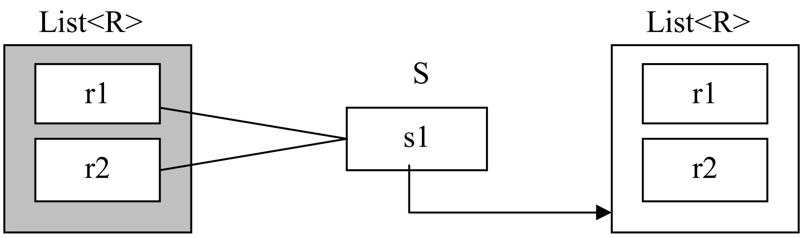
Solution-step by step

First of all, we create the root mapping for creating a new list and populate it!

```
var rootmap = mngr.RegisterMapping<List<R>>("test")
      .NewInstance().UseConverter(new ListCrossConv());
```

- `NewInstance`: We clone, so NewInstance is recommended ;-)
- `UseConverter` (…): To copy list content, we use the dedicated converter `ListCrossConv`

Leaving this will as is only creates a new List<R>, filling it with the source instance of R:



S is here, and the whole tree is inconsistent:
- New list contains old refs to {r1, r2}
- Old "s" points to the old list<R>

So we must create a sub-mapping for R:

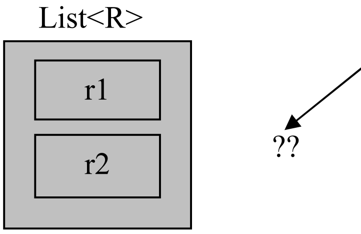
```
var RSubMap = mngr.RegisterMapping<R>()
      .NewInstance();// Create new instance if R
```

➔ We provide **`NewInstance`** to have new R instance (because we want to clone)

And provide it to the `ListCrossConv`:

```
var rootmap = mngr.RegisterMapping<List<R>>("test")
      .NewInstance().UseConverter(new ListCrossConv(RSubMap)); // sub mapping
```

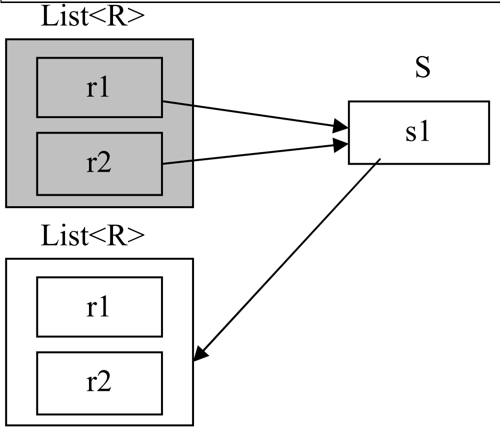
As a result:



S is out! That's logic, because the **`RSubMap`** don't provide any way to do that.
{r1, r2} are just new instances, that's all.

➔ So we need to map s:

```
var RSubMap = mngr.RegisterMapping<R>().NewInstance();
RSubMap.Map(r=>r.s); // this map "s"
```

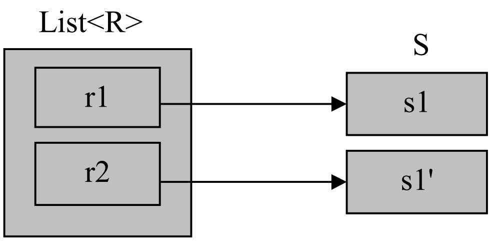


S is now here, but the whole tree is inconsistent:
- New {r1, r2} point to the old "s"
- Old "s" point to the old list<R>

➔ The reason of that is because mapping to "r.s" do not have new instance, so the old one is taken and used (FOM is an object mapper). So:

```
var RSubMap = mngr.RegisterMapping<R>().NewInstance();
RSubMap.Map(r=>r.s).NewInstance(); // map "s" and create new instance of it
```

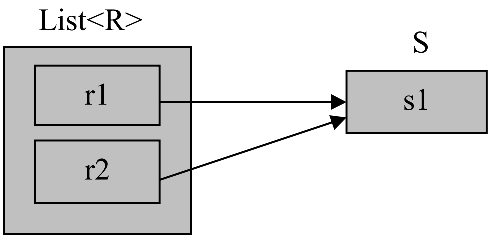
Here is the result:



S is cloned, but for each R. that logic, that's what we describe in the sub-mapping.

➔ To prevent this, we need to use and activate the new instance cache capability for s:

```
var RSubMap = mngr.RegisterMapping<R>().NewInstance();
RSubMap.Map(r=>r.s).NewInstance(true); // this maps "s" and creates a new
instance of it. Cache is used to prevent recreating a new instance of S if already done
for the same source reference.
```

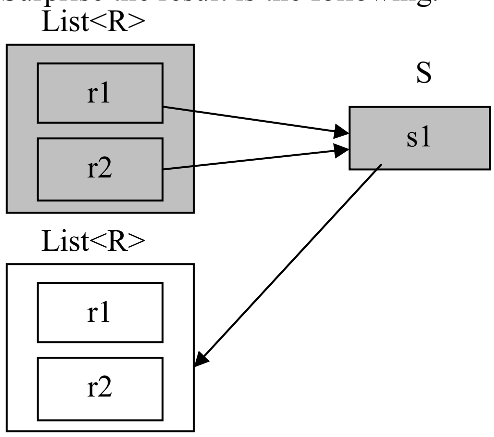


We are close to the final result. But "s1" dot not have a reference to the List<R>. This doesn't observe the 1-n relationship rule.

➔ So we need to map, from S, the List<R>:

```
var RSubMap = mngr.RegisterMapping<R>().NewInstance();
RSubMap.Map(r=>r.s).NewInstance(true)
      .Map(s=>s.rlist); // map the list of R in s
```

Surprise the result is the following:



Don't forget: FOM is an object mapper, so he maps the source to a target.

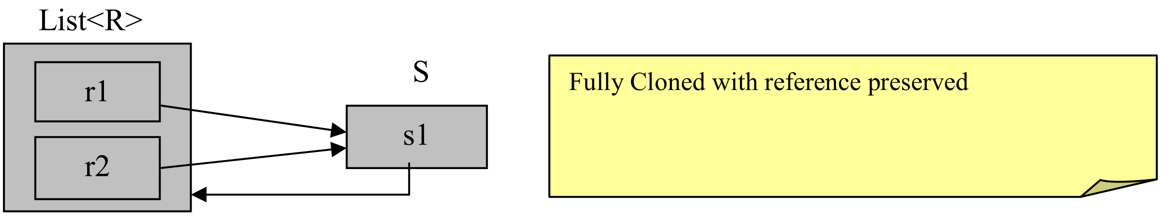
➔ We need to use the NewInstance() to have a new List<R>…. But wait! We already have one! It was created at the top of the mapping.

➔ So we must use here again the cache capability:

```
var RSubMap = mngr.RegisterMapping<R>().NewInstance();
RSubMap.Map(r=>r.s).NewInstance(true)
      .Map(s=>s.rlist).NewInstance(true); // map the list of R in s
```

➔ And we must activate it at the top of the map (this is the spec of cache using: We don't store new instances unless explicitly specified).

```
var rootmap = mngr.RegisterMapping<List<R>>("test")
.NewInstance(true).UseConverter(new ListCrossConv(RSubMap)); // sub mapping
```



Fully Cloned with reference preserved

➔ Finally, to map value types or R and S, just use the converter that can do that:

```
var RSubMap = mngr.RegisterMapping<R>().NewInstance()
      .UseConverter(new PropertyCopierConv(true));

RSubMap.Map(r=>r.s).NewInstance(true)
      .UseConverter(new PropertyCopierConv(true))
      .Map(s=>s.rlist).NewInstance(true);

var rootmap = mngr.RegisterMapping<List<R>>("test")
      .NewInstance(true).UseConverter(new ListCrossConv(RSubMap));
```

➔ Each value types will be copied from source to target.

Performance considerations

If you take a better look, each "r" instance will lead to have S and value types inside S mapped. It's logic because each "r" instance is processed independently from the others:

- r1.s1
  - r1.s1.**rlist**
  - r1.s1.(**all value types in s1**)
- r2.s1
  - r2.s1.**rlist**
  - r2.s1.(**all value types in s1**)

But because it's the same instance of "S" (s1), it time consuming to remap values already mapped. How to avoid this useless behavior?

➔ By taking advantage of cached "s" instances (again) and use a condition that will block when "s" source instance has already been processed.

```
var RSubMap = mngr.RegisterMapping<R>().NewInstance()
      .UseConverter(new PropertyCopierConv(true));

RSubMap.Map(r=>r.s).NewInstance(true)
      .UseConverter(new PropertyCopierConv(true))
      .UseCondition(new NotInCacheCond())
      .Map(s=>s.rlist).NewInstance(true);

// overload-mapping to ensure r.s if copied
RSubMap.Map(r => r.s).NewInstance(true);


var rootmap = mngr.RegisterMapping<List<R>>("test")
      .NewInstance(true).UseConverter(new ListCrossConv(RSubMap));

// run it
var ret = (List<R>)mngr.ApplyMapping("test", source);
```

➔ The fist time "s1" is processed (from r1 typically), source "s1" is not in cache. So r.s mapping is done, and then "rlist" mapping AND converter for value types are processed.

➔ The second time (from r2), the condition is false because source "s1" is already in the cache**. r.s is not processed. So neither converter nor *rlist* mapping are called.** But the second r.s mapping ensures that r.s is processed anyway (*hey! we must re-link r to s*).

➔ We have gained time (imagine a list of 100 000 R instances with 30 value types to remap!!!)

➔ 5 lines: That the final code, the most compact and efficient. Now, try to do a maintainable code doing the same thing to compare (not an ugly code I mean).

➔ Other solutions using **NotCacheCond** can be imagined to have a similar (close to) behavior.

Tip:

In the case of List<R> containing multiple same references (ex {**r1**, r2, **r1**}, just add "true":

```
var RSubMap = mngr.RegisterMapping<R>().NewInstance(true); // cache reuse
```

Absurd cases

Well, **FluentOMapper** has a wide syntax and I tried to drive the logic to the end. There are combinations that may exist and I don't have even thought about.

For example, if you write a converter that process deep actions, and further, you add a mapping that scratch your converter work, this is not a bug.

If you find an interesting case that means something but that doesn't work as expected, please write me. I will study the case.

Root mapping on value type (or string)

```
var rootmap=mngr.RegisterMapping<int>("testint");
int i=5;
int res=(int)mngr.ApplyMapping(i);
```

➔ That will return … well … **`res=5`**

➔ Only useful if using a converter that may transform your int in a complex object …

Multiple mapping on the same target

```
rootmap.map(s=>s.val1, d=>d.valX);
rootmap.map(s=>s.val2, d=>d.valX);
```

➔ As you can expect, the second one is taken in account.

Chained stupid mappings

So you can write:

```
Map(a=>a, b=>b).Map(a=>a, b=>b).Map(a=>a, b=>b).Map(a=>a, b=>b)
```

➔ I don't think this is really useful, except warming CPU. Ok, you can apply several behaviors between each, like:

```
Map(a=>a, b=>b).UseConverter(conv1).UseCondition(xxx).
.Map(a=>a, b=>b).UseConverter(conv2).Newinstance()
.Map(a=>a, b=>b).OverrideTargetType<XX>()
.Map(a=>a, b=>b);
```

➔ So, if you have special examples that you found useful, let me know. I will add it to this doc, trying to explain it.

The ultra compact writing

As soon as you'll be familiarized read FOM syntax, you will need, like me ;-), to use compact syntax.

Compact syntax is a way to present the same methods, but names are short, with 2 cars in fact. That's all:

So here they are!

Manager:

`RegisterMapping()`	➔	`RM()`
`UnRegisterMapping()`	➔	`UM()`
`GetMapping()`	➔	`GM()`
`ApplyMapping()`	➔	`AM()`

Mapping Node:

`Map()`	➔	`MP()`
`Exclude()`	➔	`EX()`
`NewInstance()`	➔	`NI()`
`GoTo()`	➔	`GT()`
`OverrideTargetType()`	➔	`TT()`
`OverrideSourceType()`	➔	`ST()`
`UseConverter()`	➔	`CV()`
`UseCondition()`	➔	`IF()`

Some examples?

Yes.

Over cases studying; you can find examples in unit test sources. I've nothing better to offer because unit tests cover more than 30 cases. It would be a heresy to duplicate them here.

Some comments exist in the unit test (with tons of orthographic errors).

Multithread consideration

Yes, a mapping call is multithread. So you can call/create multiple mapping in multiple threads. But note that giving the same source object to multiple mappings in multiple threads may lead to erratic results, mainly if your getters are not thread safe. For setters, it would be useless to alter the same property reference in two different threads, isn't it?

Performances considerations

FOM has been written to reduce the amount of treatments and memory use. But don't dream. Mapping a 2Go tree object is not a piece of cake.
Moreover, if you have Types overriding, new instances with cache, and esoteric array mappings, FOM will not do miracles. It will run as fast as the C# and the computer can.

As you can imagine, it make intensive use of c# reflection, not reputed to be the fastest mechanism.
If you find it too slow for you job, I have an only answer: Do it yourself to fit your needs and avoid useless time consuming processing.

Are there some worms or viruses in this API?

Yes! Grabbing your card number and making tons of transactions to my account to Caiman Islands. I'll soon be rich. Ha Ha Ha Ha ….. Cough … cough

Her … no. Not under my approved releases. ;-)

Bug report

Hope there is not so much …. But, in this case, please write to:
fom (at) softisis.fr

I've tried throughout Unit Tests, to cover all cases.

FFJ – Softisis 12/2011

About the author

FFJ is a Consulting Architect in Java and "most recently" Dot.net. He has been working for major French banks and a few car industries for 12 years.

He has created the site http://www.piloteo.fr to manage personal finances (French only for now). This site is a J2EE application using Hibernate, Spring, Struts2, JSP, AOP, CSS, HTML, JavaScript and AJAX technologies.