**AutoMapper**

**Pattern**:

* Mapper.CreateMap<TSource, TDestination>();
* [optionally]:

.ForMember(dest => dest.xyz, option=> .ForMember(dest => dest.xyz, option=> {

option.MapFrom(src=>src.abc);

option.ResolveUsing(...);

option.AddFormatter(...);

option.NullSubstitue(...);

});

* TDestination destInstance = Mapper.Map<TSource, TDestination>(sourceInstance);

**Validation**: Mapper.AssertConfigurationIsValid(); // throw exception and pinpoint the error location.

* 3 ways to fix: 1. Custom ValueResolver 2. Explicit Mapping 3. option => option.Ignore()

**Convention** that automagically WORKS when members have the same simple type AND:

|  |  |
| --- | --- |
| Source | Destination |
| MyProp | MyProp |
| GetMyProp | MyProp |

**Flattening**:

e.g. public class Customer { public string Name {get;set;} }

|  |  |
| --- | --- |
| Source | Destination |
| Customer Customer; | string CustomerName |

* The Source’s property has a ComplexType Customer which has its property “Name” will automagically mapped to “CustomerName” at the Destination type. By PascalCase convention.

**Explicit Mapping (Projection)**

* Mapper.CreateMap<TSource, TDestination>()
  + .ForMember(dest => dest.myProp1, opt => opt.MapFrom(src => src.memberA)
  + .ForMember(dest => dest. myProp2, opt => opt.MapFrom(src => src.memberB)
  + .ForMember(dest => dest. myProp3, opt => opt.MapFrom(src => src.memberC.myInnerProp));

**List/Array**

* Automagically support IEnumerable<T>, ICollection<T>, IList<T>, ObservableCollection<T>, Array …
* Also support polymorphic collection. i.e. collection items have both base and derived types.

**Nested Complex Type**: i.e. one complex type is a data member of another complex type.

* Both Nesting and Nested complex type needs to be mapped using their corresponding source & destination

**Global TypeConverter**: note - TDestTypeConverter implements ITypeConverter<TSource, TDest>

* Mapper.Create<TSource, TDest>().ConvertUsing(Convert.ToTDest);
* Mapper.Create<TSource, TDest>().ConvertUsing(new TDestTypeOnverter());
* Mapper.Create<TSource, TDest>().ConvertUsing<TDestTypeConverter>();
* Also support .Net System.ComponentModel.TypeConverter

**Member-level ValueConvert**

public class MyValueResolver : ValueResolver<TSource, TDestDataMember> {

protected override TDestDataMember ResolveCore(TSource source) {

return source.myProp1 + source.myProp2; // e.g. FirstName + LastName or myList.Count(); }

* Mapper.CreateMap<TS, TD>().ForMember(dest=>dest.myProp, opt=>opt.ResolveUsing<MyValueResolver>());
* Performance: (instead of using Reflection, new up the Resolver manually.)
  + .ResolveUsing<MyValueResolver>()**.ConstructedBy(()=>new MyValueResolver()**);

**ValueFormatter**

public class MyFormatter : **IValueFormatter** {

public string FormatValue(ResolutionContext context) {

// format the original value: context.DestinationValue

// to a formatted a custom string to return

}}

**NullSubstitution**

* .ForMember(dest=>dest.myProp, opt=>opt.NullSubstituion(“N/A”));

**Container**: <https://github.com/AutoMapper/AutoMapper/wiki>

**Mapping Inheritance**: <https://github.com/AutoMapper/AutoMapper/wiki>