Normal mappings

You can define multiple **mapping**s within a single binding, as already demonstrated in the last collections examples. These mappings may all be top-level (children of the **binding** element), or may be nested within other **mapping** definitions. Nested **mapping** definitions are only usable within the context of the containing **mapping**. In general, it's not a good idea to nest mapping definitions more than one level deep, or inside mappings other than the one used for the root element of your documents, because of performance concerns.

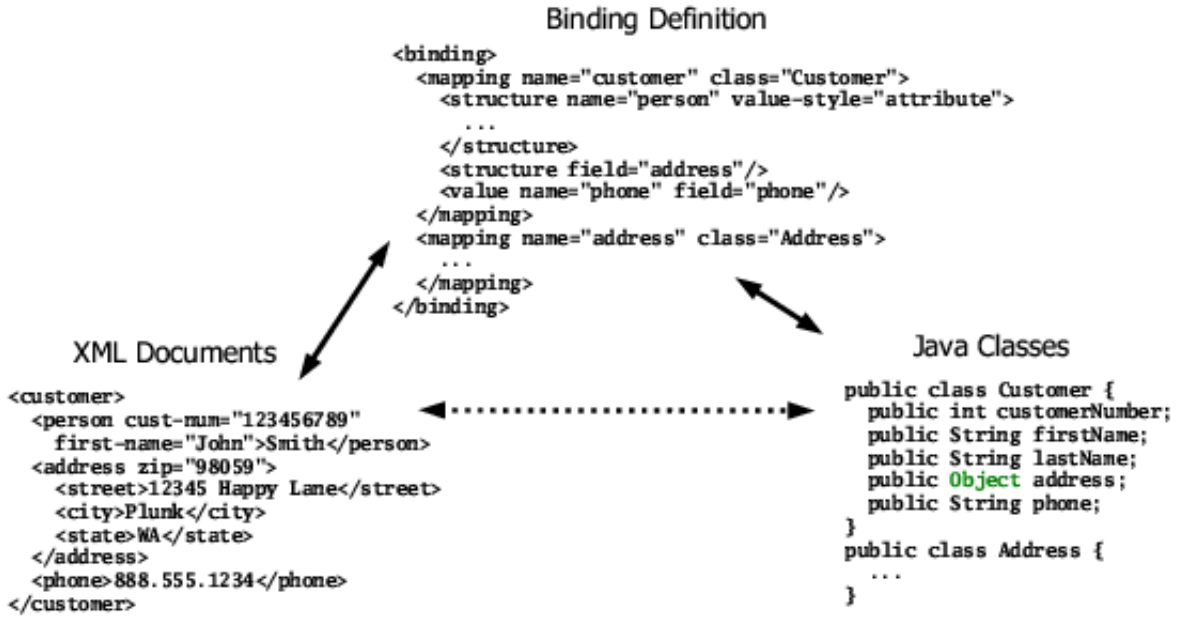
Figure 12 is a trivial example of using multiple mappings. The first **mapping** in this example binds the root **customer** element to the `Customer` class. The second **mapping**, highlighted in green, binds **address** elements to the `Address` class. The empty **structure** element (with no child elements) for the `address` field, both shown in blue, automatically uses the mapping defined for the `Address` class because that is the type of the field referenced by the **structure** element.

Figure 12. Normal multiple mapping example

Using empty **structure** elements to reference mappings, as shown in Figure 12, is most useful when an object reference can be to instances of different classes. When there's no **mapping** defined for the exact type of the object reference associated with the structure, but there are one or more mappings for assignment-compatible types (such as subclasses), JiBX will accept any of the assignment-compatible types at runtime and use the appropriate mapping.

Figure 13 demonstrates using an empty **structure** element with no **mapping** matching the object type. The only change from the last example is that I changed the type of the `address` field to `java.lang.Object` (highlighted in green). The same document is still marshalled and unmarshalled, but there is a subtle difference between the two examples: The Figure 13 binding will allow the `address` field to reference an instance of *any* mapped class, including another instance of the `Customer` class. This flexibility may not be what we want in this case (since the field is named "address"), but it fits the field definition.

Figure 13. Multiple mapping with generic reference

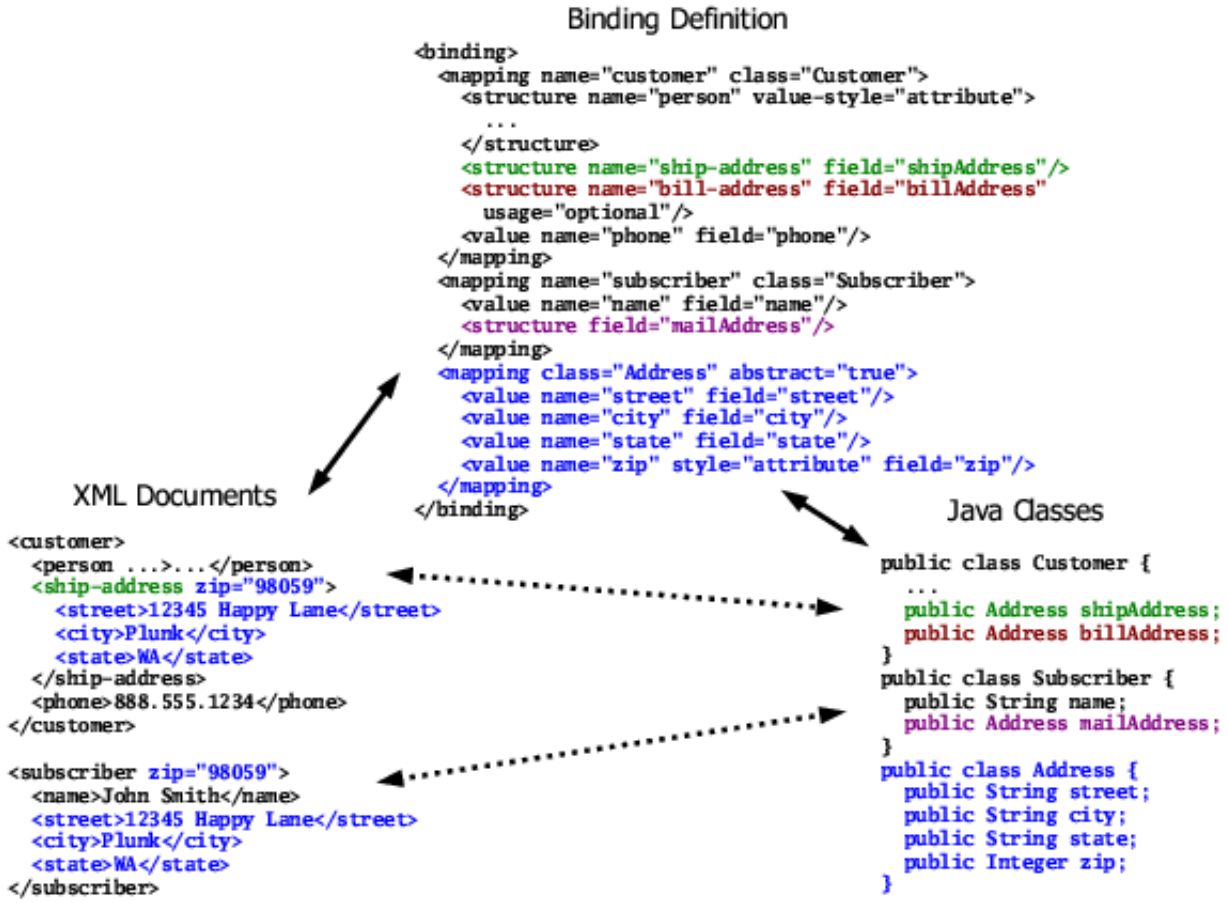


You can force an empty **structure** element to use a particular **mapping** with the **map-as** attribute. This restricts the value for the referenced object to always be of the type of that mapping. It's generally a good idea to use **map-as** when you're expecting to use a specific mapping, even when JiBX will automatically select that mapping (as in Figure 12), just to make the linkage to a particular **mapping** explicit and avoid any potential confusion. I'll show some examples of using the **map-as** attribute later on this page.

Abstract mappings

All the **mapping** examples I've used so far are normal mappings, each relating an element name to instances of a particular class. JiBX also lets you define abstract mappings, which are essentially anonymous bindings for classes. Abstract mappings can be referenced from different contexts with different element names, or with no name at all. Figure 14 shows an example using an abstract mapping.

Figure 14. Simple abstract mapping



This binding defines normal mappings for two classes, the `Customer` class and the `Subscriber` class. The abstract mapping for the `Address` class, highlighted in blue, defines the basic XML structure used to represent an address. The `Customer` mapping includes a pair of **structure** elements (highlighted in green and red) that define element names for the two `Address` components of a customer, so within the corresponding **customer** element these components are each bound to separate child elements. The `Subscriber` mapping uses a single **structure** element (highlighted in magenta) with no element name for its `Address` component, so the address information is in this case merged directly into the representation of the corresponding **subscriber** element.

You can define multiple abstract mappings for the same class, using names to distinguish between the mappings. Figure 15 shows a modified version of the last example, where I've defined two different abstract mappings for the `Address` class. The first abstract mapping, highlighted in blue, is the same as the `Address` mapping in the last example except for the addition of a **type-name** of "normal-address". The second abstract mapping, highlighted in green, uses a different structure and a **type-name** of "compact-address". Each **structure** reference to a `Address` object has also been changed, as highlighted in magenta, to reference one of the two abstract mapping names using a **map-as** attribute.

Figure 15. Named abstract mappings

The Figure 15 changes have no effect on the document with the **customer** root element, since the abstract mapping used for the `Address` instances in this context remains the same. But the document with the **subscription** root element has a very different structure from the previous example, as defined by the alternative abstract mapping for the `Address` class.

You can use interfaces as well as regular classes for abstract mappings, which is useful when the interface defines get/set methods to be used by the mapping. You can also use interfaces and abstract classes with a normal mapping definition in some circumstances - the basic requirement here is that there has to be a way to create an instance of the interface or abstract class when unmarshalling (as with a **factory** method, discussed in User extension method hooks).

mapping definitions are normally used for your own data classes, and those classes are directly modified by the JiBX binding compiler. You can also define **mapping**s for unmodifiable classes, such as system classes or ones in jar files (though you can make the classes from a jar file modifiable by just unpacking the jar before running the binding compiler, then repacking it after). When a **mapping** is defined for an unmodifiable class, the mapping can only be used within the context of some other **mapping**.

Mappings and inheritance

Besides the "free standing" mappings you've seen so far, you can also define extension mappings which are linked to other mappings. Each extension mapping references some base mapping. By attaching itself to that base mapping, the extension mapping becomes an alternative to the base mapping anywhere the base mapping is invoked in the binding. When marshalling, the actual type of the object instance determines which mapping is applied, and hence the element name (and actual representation) used in the generated XML. When unmarshalling, the element name is used to select the extension mapping to be applied and

hence the type of object to be used for the unmarshalled data.

Extension is easiest in the case where the base mapping class is never used directly. In this case you can define an abstract mapping as the root for extension, then add normal mappings which extend that abstract mapping for each substitute you want to allow in your binding. Figure 16 shows a simple example of this approach, with a base class containing some common information and a pair of subclasses providing additional details. This sort of polymorphic class hierarchy is a common use case for extension, but not the only use case.

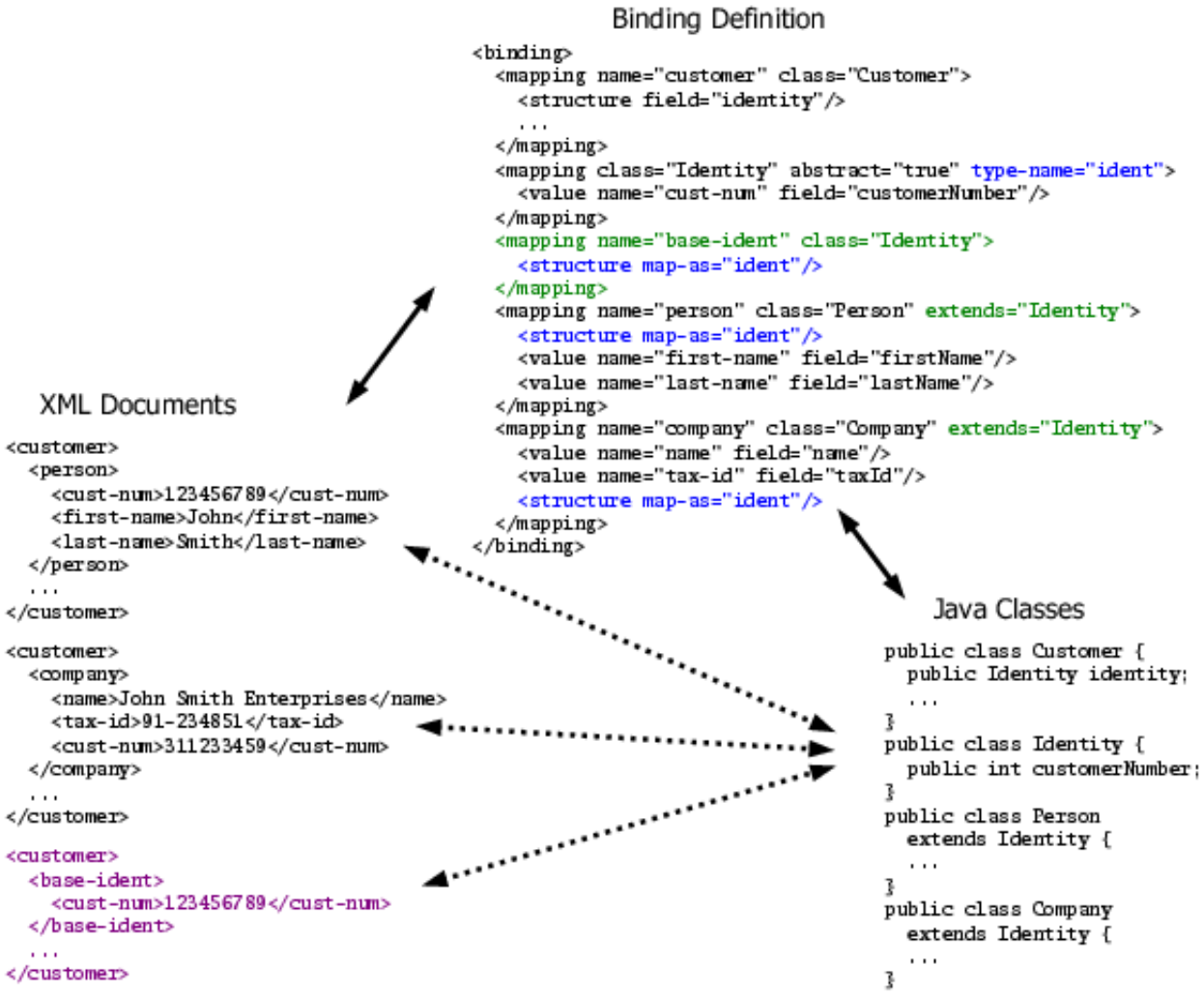
Figure 16. Abstract and extension mappings

In Figure 16 I've changed my earlier example code to support two types of customers, persons and companies. The `Customer` class doesn't care which is used, it just includes a reference to an instance of the `Identity` class. The `Identity` itself only defines a customer number. The `Person` and `Company` classes each extend `Identity` with added information for their particular type.

I've highlighted the handling of the base `Identity` class in blue, with the `Person` handling in green and the `Company` extension in magenta. The **mapping** definitions for the subclasses in this case each invoke the base class abstract mapping as part of their bindings (using a **<structure map-as="...">** reference). This is not a requirement of using extension mappings; in fact, there's no requirement that the extension mapping classes are even related to the base mapping class. Likewise, you don't need to use "extends" in order to reference a base class mapping in this way - extends is only necessary (or appropriate) in cases where instances are being used polymorphically, as in the reference from the `Customer` class in this example.

The base for an extension mapping doesn't have to be an abstract mapping. It's generally easiest to structure your extensions to use an abstract base when you can, but there are cases where that's just not possible. For example, if instances of your base mapping class can be used directly (rather than only instances of the extension mapping classes), you need to define a concrete mapping for that root class. Figure 17 gives an example of this situation, again using a set of related classes.

Figure 17. Extending a concrete mapping



The difference from the last example is that in Figure 17 the base `Identity` class can be used directly, as shown by the added example document (highlighted in magenta). Since the base class can be used directly I had to define a concrete mapping for the class, and make the subclass mappings extend the base mapping (highlighted in green). I couldn't just invoke this base class mapping in the subclass mappings, though, since the base mapping includes the **base-ident** element name - invoking it directly would correspond to an XML structure where the entire **base-ident** element was embedded within the subclass representations. Instead, I used a separate named abstract mapping to represent the structure I wanted for the base class information, and invoked this abstact mapping from both the base mapping and the subclass mappings.

Even though the original intention of extension mappings was to represent polymorphism, they can also be used in other circumstances - there's no requirement that the classes handled by the extension mappings have any particular inheritance or implements relationship to the class handled by the base mapping. This flexibility can be useful when working with XML representations which assume a particular inheritance structure (in the form of substitution groups) that doesn't match the intent of your application code.

This page covers most of the **mapping** element options and usage, but see the <mapping> element details page for full details.

Next: Advanced binding features