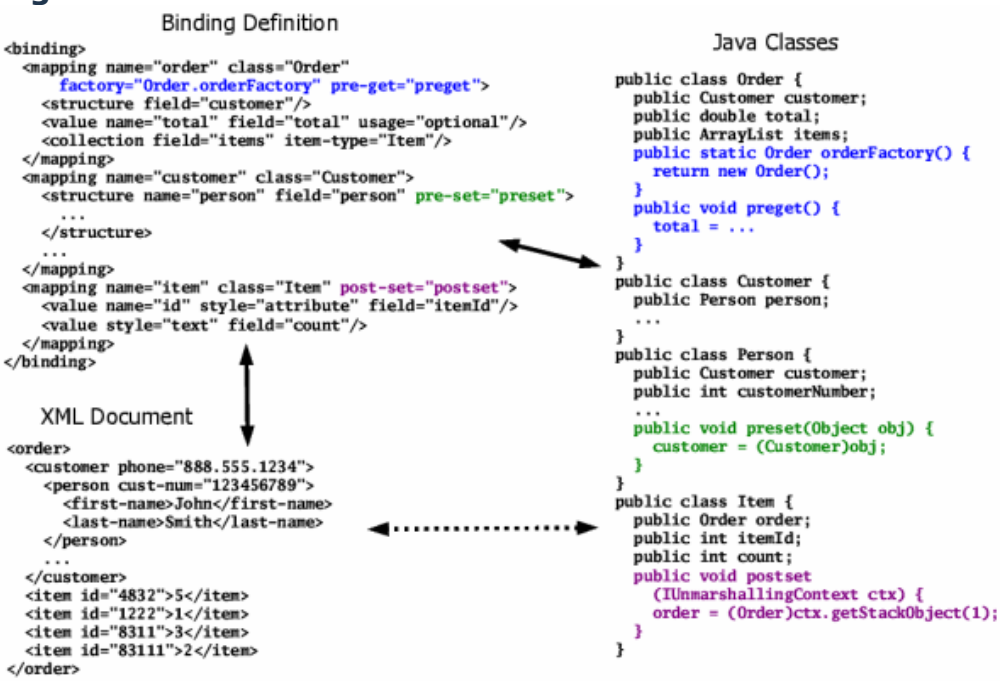
User extension method hooks

You can use your own methods in combination with JiBX marshalling and unmarshalling to support your use of the framework. Figure 19 gives examples of the various types of user method calls supported.

Figure 19. User method hooks



Starting from the top of the binding definition, I've defined both **factory** and **pre-get** methods for the `Order` class, with the relevant parts of the diagram highlighted in blue. The **factory** method is called by JiBX to create an instance of the associated class, replacing the default (no-argument) constructor JiBX otherwise expects to find for the class. A factory method needs to be static, but doesn't have to be in the class that it creates (so a fully qualified class and method name is required when you define a factory).

The pre-get method defined for the `Order` class is called by JiBX immediately before it marshals the associated object. This gives you a chance to handle any validation or completion processing you want to perform, such as calculating the order total in this case. If you use this option, the method to be called must always be a member method of the object class. Since it has to be a member method, you only need to give the method name in the definition.

For the `Person` class I've defined a **pre-set** method, highlighted in green in Figure 19.

The pre-set method is called by JiBX during unmarshalling, after an instance has been created but before any component values have been unmarshalled. This lets you do any preliminary setup you might want to perform for the object. Here again, the method must be a member method of the object class, and only the method name is used as the attribute value.

You might notice that the pre-set method example uses a different signature than the pre-get example. JiBX actually gives you three different options for the signatures of any of these extension methods. The first is a method with no arguments, as shown for the factory and pre-get examples. The second is a single argument of type `java.lang.Object`, as shown for the pre-set example. The third signature variation is

used for the last method example in Figure 19, the **post-set** method defined for the `Item` class (shown in magenta in the diagram). I'll describe the reasons for using these different signatures in a moment, but first I'll cover this last method example.

This post-set method is called by JiBX after the basic unmarshalling of the associated object (and all child objects) is complete. The only thing left to be done in the unmarshalling of the object at this point is to fill in links to forward-referenced IDs from the XML document, which won't take place until the referenced objects are unmarshalled. The post-set method is a great place to handle any special validation or linking of unmarshalled objects.

Now that that's out of the way, here's the story on the signature variations. The no-argument form of signature doesn't really require much explanation - JiBX calls your method and the method does whatever it wants, working only with the actual object (after creating the object, in the case of a factory method). If you instead use a method with a single parameter of type `java.lang.Object`, JiBX will pass the ***containing*** object as the parameter on the method call (or `null` if this is the root object being marshalled or unmarshalled). In the pre-set example of Figure 19 (shown in green) I use this variation. In the method code I take advantage of my knowledge that the containing object for a `Person` is always going to be a `Customer`, and cast the object to the latter type before saving it.

If you use a method with a single parameter of type `org.jibx.runtime.IMarshallingContext` (for a pre-get method) or `org.jibx.runtime.IUnmarshallingContext` (for the other extension methods) JiBX will pass its own context when calling your method. This can be useful for many different purposes. In Figure 19 I use this form for the post-set example method, highlighted in magenta. The reason for using the context in this case is that it gives me access to not just the containing object being unmarshalled, but the entire stack of nested objects being unmarshalled. The `Item` class is used within a collection, so the containing object of an instance -- what I'd get passed if I used an `Object` parameter in the method definition -- will be the actual `ArrayList` instance. In order to get to the `Order` object I want to access from my `Item` instance I have to use the context `getStackObject` method to go down a level in the stack.

All the extension methods can be used with non-abstract **mapping** elements, and also with **structure** and **collection** elements that work with object properties (but not with a structure that only defines an element name without an associated object).

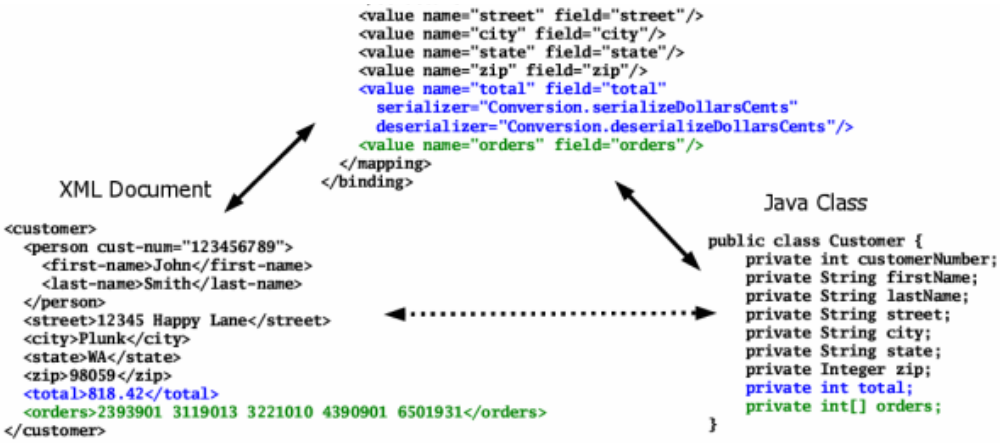
Custom serializer and deserializer methods

You can also easily use your own methods for converting values to and from string representations. Figure 20 gives an example of doing this for a couple of sample cases. The first is a dollars-and-cents value in the XML representation that converts to a `int` primitive in the Java code (giving the value in cents), shown highlighted in blue. The second is a list of numbers in the XML representation that converts to an array of `int`s in the Java code, shown highlighted in green.

Figure 20. Using custom serializers and deserializers

Binding Definition

```
<binding>
  <format type="int[]" serializer="Conversion.serializeIntArray"
    deserializer="Conversion.deserializeIntArray"/>
  <mapping name="customer" class="Customer">
    <structure name="person">
      <value name="cust-num" style="attribute"
        field="customerNumber"/>
      <value name="first-name" field="firstName"/>
      <value name="last-name" field="lastName"/>
    </structure>
```



This shows two ways of defining custom conversions. The first, used for the dollars-and-cents values (highlighted in blue), associates the serializer and deserializer directly with the value to be converted by using the **serializer** and **deserializer** attributes of the **value** element. The second, used for the orders list (highlighted in green), sets custom defaults for handling a particular type of value by using the **format** element. This custom default then applies to all values of the specified type within the context of the definition. In the case of the Figure 20 example, this context is the entire binding definition, since the **format** element is a child of the **binding** element.

The third way of defining custom conversions (not shown in the Figure 20 example) also uses the **format** element, but with the addition of a **label** attribute value. In this case the format does not become a default, but can be referenced by name wherever needed (using the **format** attribute of a **value** element).

The methods used for custom conversions to and from XML just need to be static methods that take a single argument and return a value of the appropriate type. For the serializer method, the argument must be of the Java type being handled (or a superclass) and the return must be a `java.lang.String`. For the deserializer method, the argument must be a `java.lang.String` and the return must be of the Java type being handled. Here's what these might look like for the Figure 20 binding:

```
public static String serializeDollarsCents(int cents) {
    StringBuffer buff = new StringBuffer();
    buff.append(cents / 100);
    int extra = cents % 100;
    if (extra != 0) {
        buff.append('.');
        if (extra < 10) {
            buff.append('0');
        }
        buff.append(extra);
    }
    return buff.toString();
}

public static int deserializeDollarsCents(String text) {
    if (text == null) {
        return 0;
    } else {
        int split = text.indexOf('.');
        int cents = 0;
        if (split > 0) {
```

```
            cents = Integer.parseInt(text.substring(0, split)) * 100;
            text = text.substring(split+1);
        }
        return cents + Integer.parseInt(text);
    }
}

public static String serializeIntArray(int[] values) {
    StringBuffer buff = new StringBuffer();

    for (int i = 0; i < values.length; i++) {
        if (i > 0) {
            buff.append(' ');
        }
        buff.append(values[i]);
    }
    return buff.toString();
}

private static int[] resizeArray(int[] array, int size) {
    int[] copy = new int[size];
    System.arraycopy(array, 0, copy, 0, Math.min(array.length, size));
    return copy;
}

public static int[] deserializeIntArray(String text) {
    if (text == null) {
        return new int[0];
    } else {
        int split = 0;
        text = text.trim();
        int fill = 0;
        int[] values = new int[10];
        while (split < text.length()) {
            int base = split;
            split = text.indexOf(' ', split);
            if (split < 0) {
                split = text.length();
            }
            int value = Integer.parseInt(text.substring(base, split));
            if (fill >= values.length) {
                values = resizeArray(values, values.length*2);
            }
            values[fill++] = value;
            while (split < text.length() && text.charAt(++split) == ' ');
        }
        return resizeArray(values, fill);
    }
}
```

When an optional value is missing in the input document the JiBX binding code will call the deserialize method with `null`, since there's no text value to pass. Calling the deserialize method allows this method to handle the case of a missing value in whatever manner is appropriate. In the Figure 20 binding the deserialize methods are only used for required

values, so strictly speaking the above implementations would not need to handle the `null` case. It's often best to code for this case anyway, though, in case new uses of the

case. It's often best to code for this case anyway, though, in case new uses of the deserializers are added in the future.

Next: Customizing JiBX binding behavior