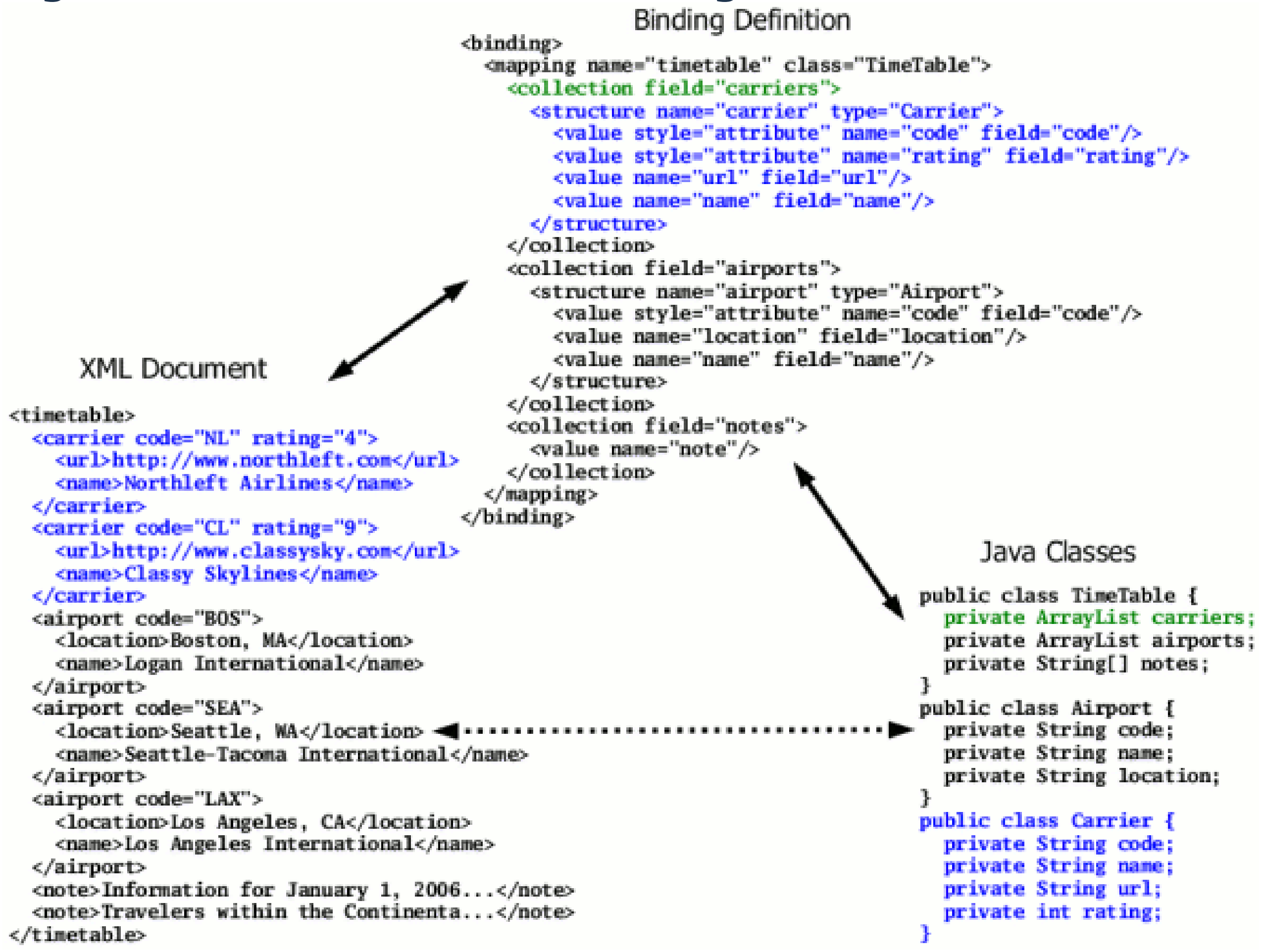
Working with collections and arrays

Besides working with individual objects, applications often need to deal with collections of objects. Figure 8 gives a simple example of using collections with JiBX. Here the classes represent the basics of an airline flight timetable, which I'll expand on for the next examples. In this example I'm using three collections in the root `TimeTable` object, representing carriers (airlines), airports, and notes.

Figure 8. Basic collection handling



```
<binding>
  <mapping name="timetable" class="TimeTable">
    <collection field="carriers">
      <structure name="carrier" type="Carrier">
        <value style="attribute" name="code" field="code"/>
        <value style="attribute" name="rating" field="rating"/>
        <value name="url" field="url"/>
        <value name="name" field="name"/>
      </structure>
    </collection>
    <collection field="airports">
      <structure name="airport" type="Airport">
        <value style="attribute" name="code" field="code"/>
        <value name="location" field="location"/>
        <value name="name" field="name"/>
      </structure>
    </collection>
    <collection field="notes">
      <value name="note"/>
    </collection>
  </mapping>
</binding>
```

```
<timetable>
  <carrier code="NL" rating="4">
    <url>http://www.northleft.com</url>
    <name>Northleft Airlines</name>
  </carrier>
  <carrier code="CL" rating="9">
    <url>http://www.classysky.com</url>
    <name>Classy Skylines</name>
  </carrier>
  <airport code="BOS">
    <location>Boston, MA</location>
    <name>Logan International</name>
  </airport>
  <airport code="SEA">
    <location>Seattle, WA</location>
    <name>Seattle-Tacoma International</name>
  </airport>
  <airport code="LAX">
    <location>Los Angeles, CA</location>
    <name>Los Angeles International</name>
  </airport>
  <note>Information for January 1, 2006...</note>
  <note>Travelers within the Continenta...</note>
</timetable>
```

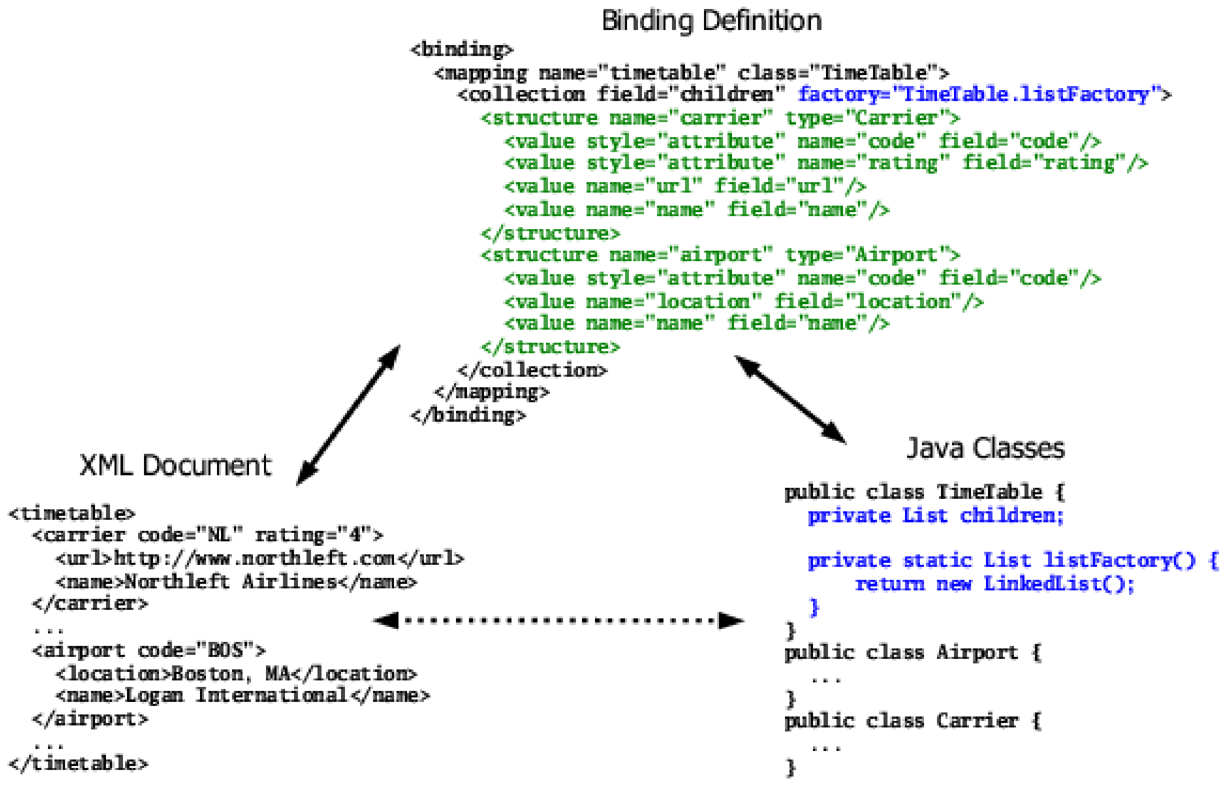
```
public class TimeTable {
  private ArrayList carriers;
  private ArrayList airports;
  private String[] notes;
}
public class Airport {
  private String code;
  private String name;
  private String location;
}
public class Carrier {
  private String code;
  private String name;
  private String url;
  private int rating;
}
```

The Figure 8 binding definition uses a **collection** element for each collection. In the case of the first two collections there's a nested **structure** element to provide the details of the items present in the collection. I've highlighted the definitions for the collection of carriers in green, and the actual carrier information in blue, to emphasize the connection between the different components. The collection of airports is handled in the same way as the collection of carriers.

The collection of notes differs from the other collections both in that it's stored as an array, and that the values are simple `String`s rather than objects with their own structure. You can use arrays of both object and primitive types with the **collection** element. In the case of simple values (primitives, or objects which are represented as simple text strings - including `String`s, as in this example), you just use one or more nested **value** elements (which must use **style="element"**, directly or by default) instead of **structure** elements. The same applies for non-array collections of simple values.

In the case of the Figure 8 binding the collections are homogeneous, with all items in each collection of a particular type. You can also define heterogeneous collections, consisting of several types of items, by just including more than one **structure** (or **value**) element as a child of the **collection** element. Figure 9 demonstrates using a heterogeneous collection for the carrier and airport data from Figure 8, with the **structure** definitions for the carrier and airport components (shown in green) combined in a single collection.

Figure 9. Heterogeneous collection, with factory



```
<binding>
  <mapping name="timetable" class="TimeTable">
    <collection field="children" factory="TimeTable.listFactory">
      <structure name="carrier" type="Carrier">
        <value style="attribute" name="code" field="code"/>
        <value style="attribute" name="rating" field="rating"/>
        <value name="url" field="url"/>
        <value name="name" field="name"/>
      </structure>
      <structure name="airport" type="Airport">
        <value style="attribute" name="code" field="code"/>
        <value name="location" field="location"/>
        <value name="name" field="name"/>
      </structure>
    </collection>
  </mapping>
</binding>
```

```
<timetable>
  <carrier code="NL" rating="4">
    <url>http://www.northleft.com</url>
    <name>Northleft Airlines</name>
  </carrier>
  ...
  <airport code="BOS">
    <location>Boston, MA</location>
    <name>Logan International</name>
  </airport>
  ...
</timetable>
```

```
public class TimeTable {
  private List children;

  private static List listFactory() {
      return new LinkedList();
  }
}
public class Airport {
  ...
}
public class Carrier {
  ...
}
```

Figure 9 also demonstrates one way to work with collection interfaces (shown in blue). I've changed the type of the collection field in the `TimeTable` class to the `List` interface, rather than the concrete `ArrayList` class used in Figure 8. I've also added the `listFactory()` method, which returns an instance of a `List` interface. Finally, I added a **factory** attribute on the binding definition **collection** element to specify the factory method. When a factory method is given, JiBX calls that method to get a new instance of the class if it needs one during unmarshalling (see User extension method hooks for full details on this and other ways of using your own code with JiBX). JiBX will reuse an existing instance if one is already present, so the method is only called if the current value of the field is `null` (though when reusing a collection, you need to "manually" empty the collection before unmarshalling - a **pre-set** method works well for this purpose).

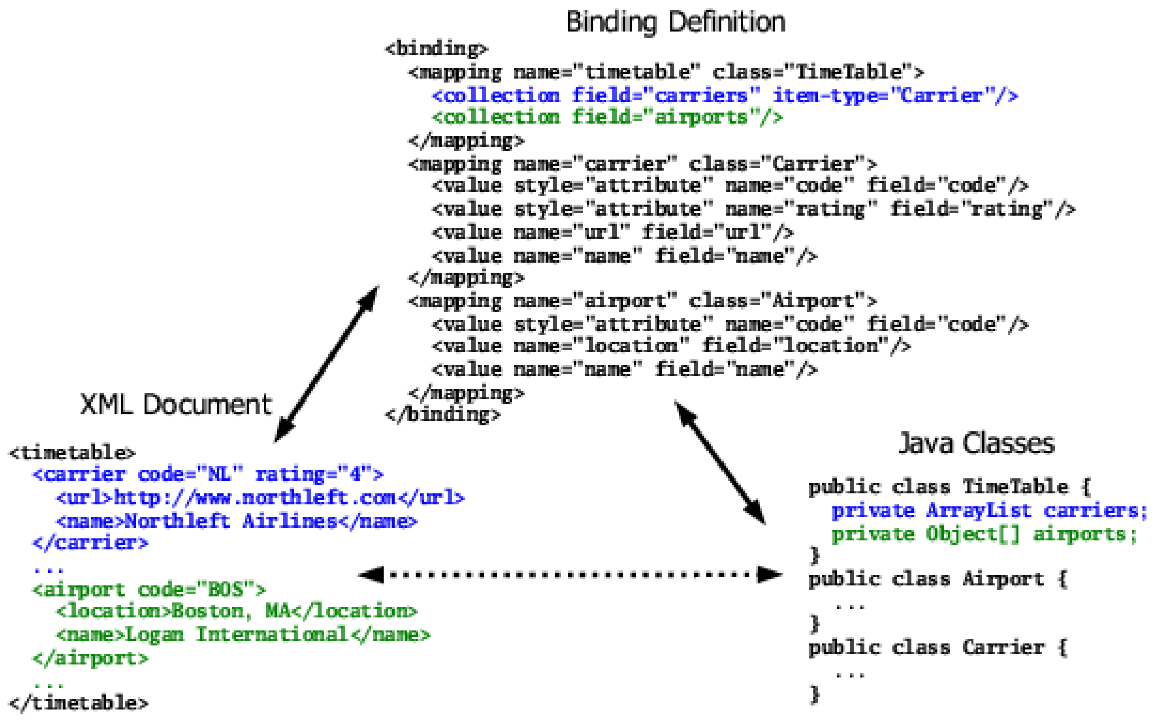
As of the JiBX 1.1 release there's an easier way to accomplish the same effect as using a **factory** to supply instances of an implementation class. This is to use the new **create-type** attribute to specify the class used when creating new instances of an object. See the object attribute group descriptions for the full details.

As with **structure** elements with multiple child components, heterogeneous collections can be either ordered (meaning the items of each type may be repeated, but the different types of items must always occur in the specified order) or unordered (meaning the items can be in any order). Either way, the child components of a collection are always treated as optional by JiBX (so zero or more instances are accepted).

The **collection** element is generally similar to the **structure** element in usage and options, but accepts some additional attributes that are unique to working with collections of items. Most of the added attributes are for when you want to implement a custom form of collection, using your own methods to add and retrieve items in the collection. Another attribute, **item-type**, can be used to specify the type of items in the collection.

For the prior examples I've used embedded **structure** elements to define the structure of items in the collection. This isn't the only way to use collections, though. You can instead leave a **collection** element empty to tell the binding compiler that objects in the collection will have their own **mapping** definitions. Specifying the type of items can be useful in this case to avoid ambiguity. Figure 10 shows an example of using mapping definitions in this way.

Figure 10. Collections with mappings



```
<binding>
  <mapping name="timetable" class="TimeTable">
    <collection field="carriers" item-type="Carrier"/>
    <collection field="airports"/>
  </mapping>
  <mapping name="carrier" class="Carrier">
    <value style="attribute" name="code" field="code"/>
    <value style="attribute" name="rating" field="rating"/>
    <value name="url" field="url"/>
    <value name="name" field="name"/>
  </mapping>
  <mapping name="airport" class="Airport">
    <value style="attribute" name="code" field="code"/>
    <value name="location" field="location"/>
    <value name="name" field="name"/>
  </mapping>
</binding>
```

```
<timetable>
  <carrier code="NL" rating="4">
    <url>http://www.northleft.com</url>
    <name>Northleft Airlines</name>
  </carrier>
  ...
  <airport code="BOS">
    <location>Boston, MA</location>
    <name>Logan International</name>
  </airport>
  ...
</timetable>
```

```
public class TimeTable {
  private ArrayList carriers;
  private Object[] airports;
}
public class Airport {
  ...
}
public class Carrier {
  ...
}
```

In Figure 10 I've converted the embedded carrier and airport **structure** definitions used in the earlier examples into their own **mapping** elements. The binding uses an **item-type** attribute to specify that the first collection (shown in blue) contains only carriers, while the second collection (shown in green) uses a generic `Object` array for the airport information. In this example, if I didn't specify the type of items present in the first collection JiBX wouldn't know when to stop adding unmarshalled items to the first collection and start adding them to the second collection. Using the **item-type** attribute makes it clear that the first collection is only intended for `Carrier` instances. I could also use an **item-type** attribute on the second collection, if I wanted to, but in this case it's unnecessary - the only thing following the carrier information in the XML representation is the airport information.

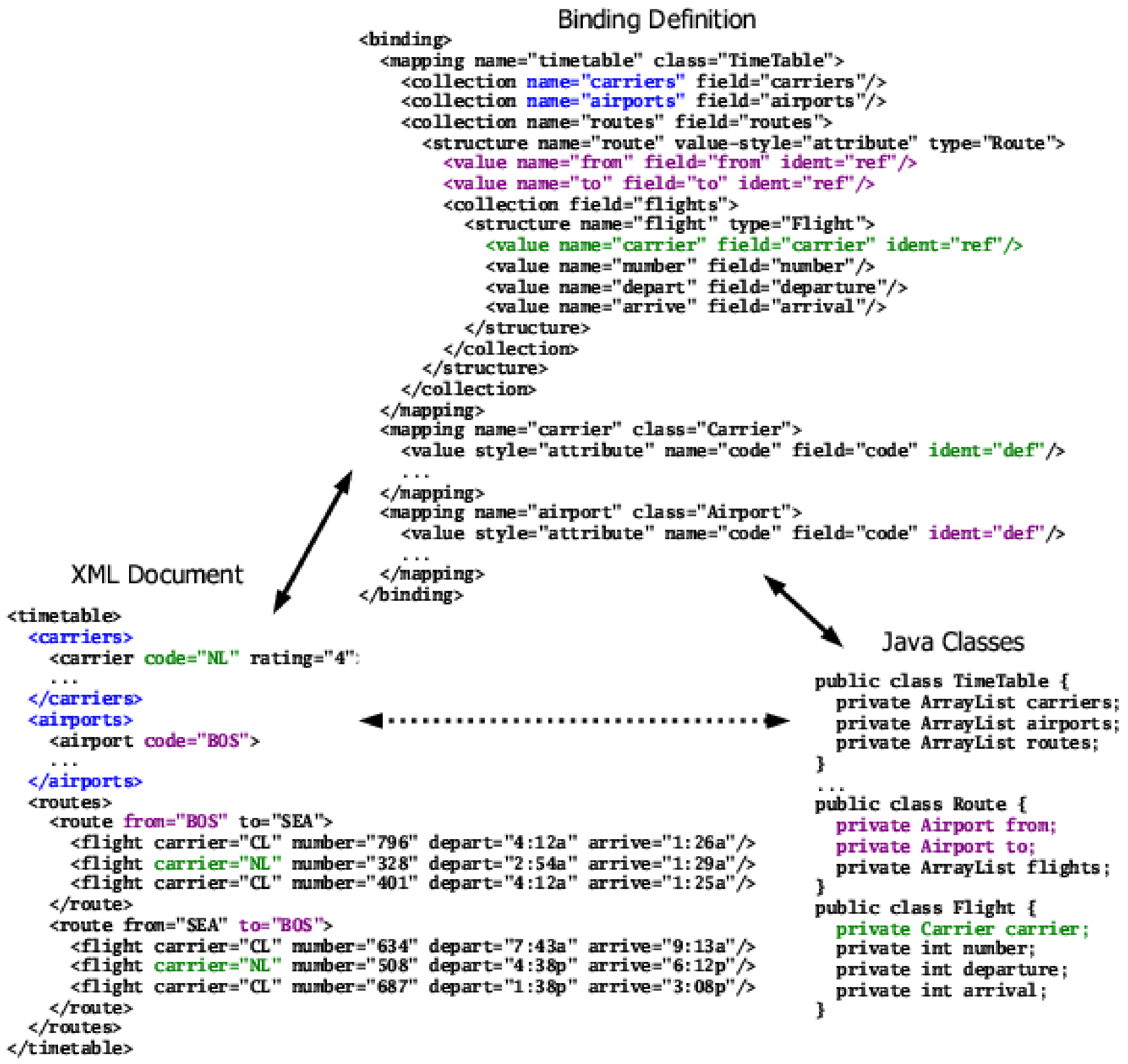
You can nest collections inside other collections. This is the approach used to represent multidimensional arrays, or Java collections made up of other collections. You can also use **value** elements directly as the child of a **collection** element, though only if the **value** representation is as an element. This is the way you'd handle a collection of simple `String` values, or an array of `int` values, for instance.

The **collection** element will work directly with all standard Java collections implementing the `java.util.Collection` interface, as well as with arrays of both object and primitive types. It can also be used with your own specialized collection types, using optional attributes to specify the methods used by JiBX to access the collection data. When used with arrays, the defined type of the array is assumed as the type of the items in the collection. You can override this to a more specific type by using the **item-type** attribute with an array. See the <collection> element details page for more details on the collection options and usage.

Working with IDs

Figure 11 gives a more complex example of working with collections. This builds on the Figure 10 XML and data structures. The prior collections of **carrier** and **airport** elements are still present, but now the XML representation uses wrapper elements (**carriers** and **airports**, respectively) for the collections of each type. The blue highlighting in the diagram shows this change. In the binding definition, the addition of the wrapper element is shown by just adding a **name** attribute to each **collection** element.

Figure 11. Collections and IDs



```
<binding>
  <mapping name="timetable" class="TimeTable">
    <collection name="carriers" field="carriers"/>
    <collection name="airports" field="airports"/>
    <collection name="routes" field="routes">
      <structure name="route" value-style="attribute" type="Route">
        <value name="from" field="from" ident="ref"/>
        <value name="to" field="to" ident="ref"/>
        <collection field="flights">
          <structure name="flight" type="Flight">
            <value name="carrier" field="carrier" ident="ref"/>
            <value name="number" field="number"/>
            <value name="depart" field="departure"/>
            <value name="arrive" field="arrival"/>
          </structure>
        </collection>
      </structure>
    </collection>
  </mapping>
  <mapping name="carrier" class="Carrier">
    <value style="attribute" name="code" field="code" ident="def"/>
    ...
  </mapping>
  <mapping name="airport" class="Airport">
    <value style="attribute" name="code" field="code" ident="def"/>
    ...
  </mapping>
</binding>
```

```
<timetable>
  <carriers>
    <carrier code="NL" rating="4"
    ...
  </carriers>
  <airports>
    <airport code="BOS">
    ...
  </airports>
  <routes>
    <route from="BOS" to="SEA">
      <flight carrier="CL" number="796" depart="4:12a" arrive="1:26a"/>
      <flight carrier="NL" number="328" depart="2:54a" arrive="1:29a"/>
      <flight carrier="CL" number="401" depart="4:12a" arrive="1:25a"/>
    </route>
    <route from="SEA" to="BOS">
      <flight carrier="CL" number="634" depart="7:43a" arrive="9:13a"/>
      <flight carrier="NL" number="508" depart="4:38p" arrive="6:12p"/>
      <flight carrier="CL" number="687" depart="1:38p" arrive="3:08p"/>
    </route>
  </routes>
</timetable>
```

```
public class TimeTable {
  private ArrayList carriers;
  private ArrayList airports;
  private ArrayList routes;
}
...
public class Route {
  private Airport from;
  private Airport to;
  private ArrayList flights;
}
public class Flight {
  private Carrier carrier;
  private int number;
  private int departure;
  private int arrival;
}
```

I've also added route and flight information to the Figure 11 binding. The most interesting part about these additions is the use of references back to the airport and carrier information. The carrier reference linkages are highlighted in green, the airport linkages in magenta. In the Java code, the linkages are direct object references. On the XML side, these are converted into ID and IDREF links - each carrier or airport defines an ID value, which is then referenced by flight or route elements. The binding definition shows these linkages through the use of an **ident="def"** attribute on the child **value** component of a **mapping** element supplying the ID, and an **ident="ref"** attribute on an IDREF **value** component that references an ID.

Using ID and IDREF links allows references between objects to be marshalled and unmarshalled, but is subject to some limitations. Each object with an ID must have a **mapping** in the binding. The current

JiBX code also requires that you define objects in some consistent way, though the references to the objects can be from anywhere (even before the actual definitions of the objects). In other words, you have to define each object once and only once. In Figure 11 the definitions occur in the **carriers** and **airports** collections. The current code also prohibits using IDREF values directly within a collection (so the definitions can be from a collection, but not the references) - to use references in a collection you need to define some sort of wrapper object that actually holds the reference. However, see JiBX extras for some support classes which extend the basic JiBX handling in these areas.

Next: The many flavors of mappings