Inheriting Java Types

The introduction of Java syntax to BBj® gave developers a vast library of existing code to add to their toolboxes. Custom Objects further offered the ability to transform the way developers write code. BBj now bridges the gap between BBj and Java by allowing developers to extend Java classes and implement Java interfaces with BBj CustomObjects.

Why?

When using object-oriented programming principles, it is not uncommon to encounter a class that fulfills almost all the requirements for a particular need. "If only I had access to the code, I could change it!" or "With just a minor tweak, I could use this class!" Other times, a class provides some base functionality but leaves it up to the developer to implement more specific details. In fact, one of the core principles of object-oriented design is that classes can be extended to tune functionality and implement more specific behavior.

When considering using classes implemented in Java, a BBj developer traditionally had to weigh some considerable costs, however. On one hand, a library may already exist to provide a significant amount of desired functionality. On the other hand, modifying or implementing any specialized behavior required learning the Java development and deployment model or contracting or even hiring a Java developer to use this existing functionality. Although it exists and might even be freely available and maddeningly accessible, the hidden costs to use a Java library would begin to reveal themselves.

What if it were not necessary to hire a Java developer or learn a new development model? What if a BBj developer could simply read the Java documentation and code their changes in BBj? Of course, the answer is that by allowing BBj CustomObjects both to extend Java classes and implement Java interfaces, these scenarios are now possible.

By Adam Hawthorne
Software Engineer

To download the code samples cited in this article, click here.

Interfaces

Some third party Java libraries specify an interface that an object must implement in order to provide some functionality. One such library is the Google Collections Library. The **`Predicate.src`** example in **Figure 1** is derived from the downloaded code. It uses the 1.0 RC1 version of the Google Collections Library to illustrate implementing a third party interface.

The **`Predicate`** interface has various uses in the Google Collections Library, but one use is creating a view of a **`java.util.Map`** that only returns values that match the given predicate (see **Figure 2**).

Notice that the sample in **Figure 2** does not reference any Java source files or Java class files for the new functionality. The BBj source code files simply uses what the Java library already provides. The developer only needs to add the JAR file for the Google Collections Library to the BBj Services classpath. This may even be done at runtime in BBj Services using the new features of SSCP, also featured in this edition of the Advantage.

While BBj 9.0 supports the sample **`Predicate.src`**, BBj 9.01 and higher supports it with a more refined and robust implementation.

Classes

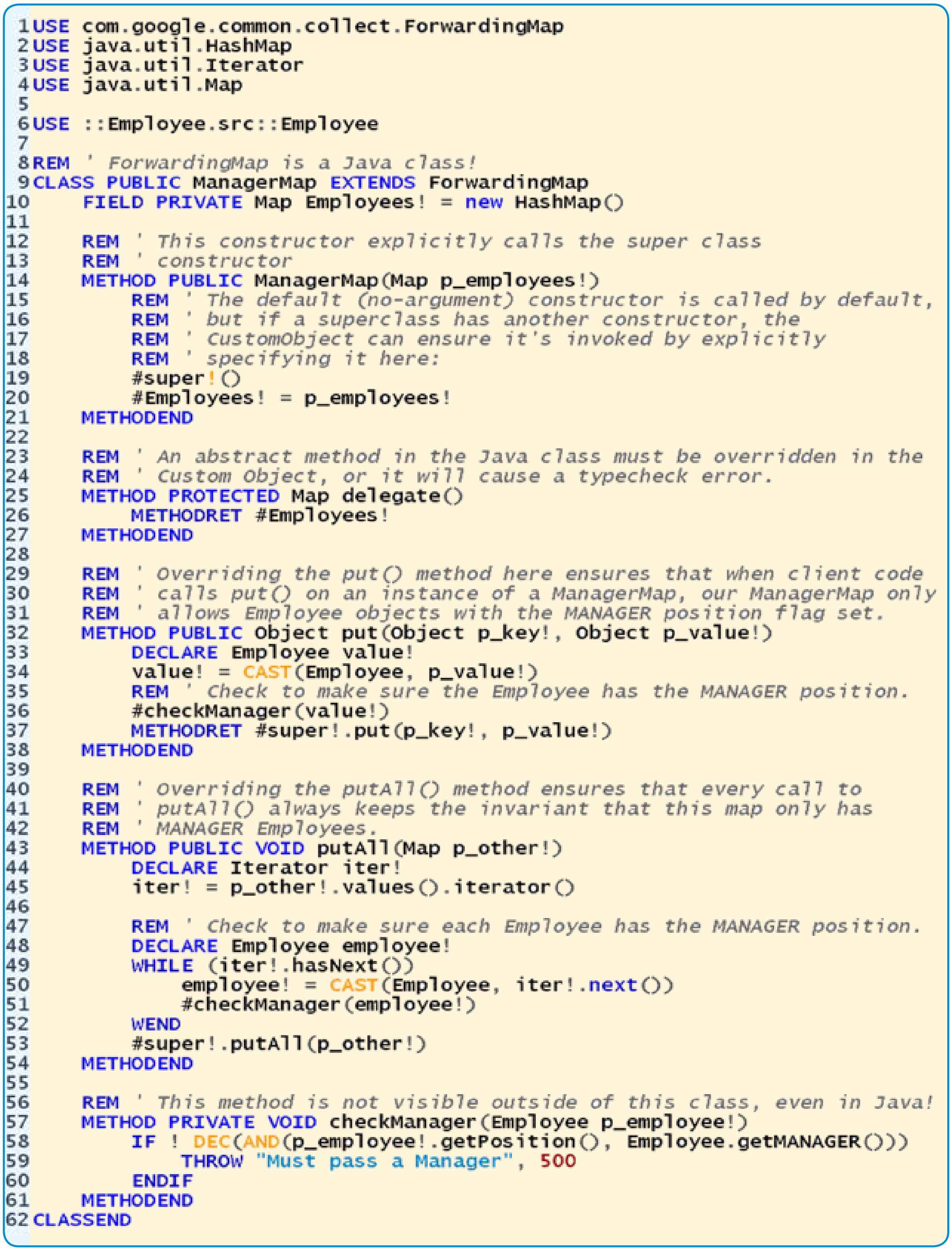
BBj 10.0 further enhances the features introduced in BBj 9.0 with the ability to inherit from Java classes as well as from Java interfaces. The ability to extend Java classes allows the developer to take advantage of an even greater collection of Java libraries. For example, XML is increasingly pervasive and is becoming more and more difficult to ignore. The Java Runtime Environment provides a rich framework to provide access to XML data, validation and other XML-based technologies, but, for example, writing a SAX parser requires a developer to extend a Java class. Before illustrating the power of this feature with the XML libraries, though, **`ManagerMap.src`** in **Figure 3** again uses the Google Collections Library to demonstrate many of the features of extending a Java Class with Custom Objects. As with the example for Java interfaces, extending a Java class with BBj Custom Objects is simple. >>

```
REM ' USE statements
USE com.google.common.base.Predicate

REM ' This class implements the com.google.common.base.Predicate interface
REM ' simply by referring to it in the IMPLEMENTS clause below and
REM ' defining the required method apply()
CLASS PUBLIC PositionPredicate IMPLEMENTS Predicate

    REM ' apply() is specified by the Predicate interface.
    METHOD PUBLIC boolean apply(Object p_employee!)
        DECLARE BBjNumber result
        REM ' Assign to result and return
        REM ' ...

        METHODRET result
    METHODEND
CLASSEND
```

Figure 1. Implementing the third party Predicate interface

```
USE com.google.common.base.Predicate
USE com.google.common.collect.Maps
USE java.util.Map

DECLARE Map employees!
DECLARE Map managers!
DECLARE Predicate mgrPredicate!

REM ' Define mgrPredicate! and get Map of employees!
REM ' ...

REM ' Use the Predicate to define a view of the employees! Map
managers! = Maps.filterValues(employees!, mgrPredicate!)
```

Figure 2. Using the Predicate interface



```
USE com.google.common.collect.ForwardingMap
USE java.util.HashMap
USE java.util.Iterator
USE java.util.Map

USE ::Employee.src::Employee

REM ' ForwardingMap is a Java class!
CLASS PUBLIC ManagerMap EXTENDS ForwardingMap
    FIELD PRIVATE Map Employees! = new HashMap()

    REM ' This constructor explicitly calls the super class
    REM ' constructor
    METHOD PUBLIC ManagerMap(Map p_employees!)
        REM ' The default (no-argument) constructor is called by default,
        REM ' but if a superclass has another constructor, the
        REM ' CustomObject can ensure it's invoked by explicitly
        REM ' specifying it here:
        #super!()
        #Employees! = p_employees!
    METHODEND

    REM ' An abstract method in the Java class must be overridden in the
    REM ' Custom Object, or it will cause a typecheck error.
    METHOD PROTECTED Map delegate()
        METHODRET #Employees!
    METHODEND

    REM ' Overriding the put() method here ensures that when client code
    REM ' calls put() on an instance of a ManagerMap, our ManagerMap only
    REM ' allows Employee objects with the MANAGER position flag set.
    METHOD PUBLIC Object put(Object p_key!, Object p_value!)
        DECLARE Employee value!
        value! = CAST(Employee, p_value!)
        REM ' Check to make sure the Employee has the MANAGER position.
        #checkManager(value!)
        METHODRET #super!.put(p_key!, p_value!)
    METHODEND

    REM ' Overriding the putAll() method ensures that every call to
    REM ' putAll() always keeps the invariant that this map only has
    REM ' MANAGER Employees.
    METHOD PUBLIC VOID putAll(Map p_other!)
        DECLARE Iterator iter!
        iter! = p_other!.values().iterator()

        REM ' Check to make sure each Employee has the MANAGER position.
        DECLARE Employee employee!
        WHILE (iter!.hasNext())
            employee! = CAST(Employee, iter!.next())
            #checkManager(employee!)
        WEND
        #super!.putAll(p_other!)
    METHODEND

    REM ' This method is not visible outside of this class, even in Java!
    METHOD PRIVATE VOID checkManager(Employee p_employee!)
        IF ! DEC(AND(p_employee!.getPosition(), Employee.getMANAGER()))
            THROW "Must pass a Manager", 500
        ENDIF
    METHODEND
CLASSEND
```

Figure 3. Specifying a Java class after the EXTENDS keyword on a line with a class declaration

>>

To run **`ManagerMap.src`** and the remaining examples in this article on BBj 9.x or lower, first run **`Pre10Setup.src`**.

Line 19 in Figure 3 shows how to invoke the superclass constructor, identical to invoking a Custom Object superclass constructor. Line 37 and line 53 similarly invoke a superclass method exactly as one would in a Custom Object. All the methods of the library class **`ForwardingMap`** call the **`delegate()`** method to find the Map delegate before calling the same method on it. Each invocation of **`delegate()`** on an instance of **`ManagerMap`** will invoke the **`delegate()`** method in **`ManagerMap`**.

Caveats

Interoperability with Java objects does provide significant functionality, but with that functionality, there a few things to watch out for.

Object Methods

Because Custom Objects are now full-fledged Java objects, the methods declared on **`java.lang.Object`** are also available to CustomObject classes. Any of the methods **`hashCode()`**, **`getClass()`**, **`equals(Object)`**, **`toString()`**, **`wait()`**, **`wait(long)`**, **`clone()`** and **`notify()`** are now implicitly defined on Custom Objects. A runtime error will occur when defining a method that has the same name and parameter types as one of these methods but that also has a different return type. Attempting to provide an implementation in a Custom Object class for a method declared as final will also cause a runtime error. Run **`bbjcpl -t`** on the source file to reveal these incompatibilities. To fix an instance of these problems, change the Custom Object method name or add a dummy argument to the method and all its invocations.

Runtime Modifications

Modifying Custom Object class structure at runtime has always been discouraged, however, the ability to inherit Java types adds another reason to avoid modifying Custom Object types at runtime. The Java runtime does not permit certain kinds of runtime modifications to Java classes. Since Custom Objects now may now have a Java component, changing the structure of a Custom Object at runtime can invalidate its contents. In that case, Custom Objects placed in Java data structures will become invalid and will cause runtime errors for any method invoked on them. Also, BBj will not preserve object identity of the Custom Object Java component through a class structure change.

Generics

Many Java APIs use Java Generics, including the JRE itself and the running example of the Google Collections Library. Generics are an extension to the type system to allow a programmer to place further constraints on the types of variables. Although they should be supported in a future version of BBj, follow the guidelines below to ensure intended behavior in the interim.

In the Java classes example above, the Java declaration for **`Map`** is **`Map<K, V>`**, which gives the developer the ability to limit the types of the keys and values respectively. **`K`** and **`V`** are called "type variables". They are used throughout the rest of the API as placeholders for the constrained types provided by the developer. For example, **`Map.put()`** uses the **`K`** and **`V`** type variables **`V put(K key, V value)`** to ensure that the map only contains keys and values of the appropriate types. However, all this work is done by the Java compiler. At runtime in Java, some of the information is unused and some of it is unavailable, in a process called "erasure". For the time being, BBj programs need to specify the "erased type" of a generic API. There are three basic rules:

1. When a Java API uses a type variable declared simply as **`T`**, use **`Object`** in any overridden method instead of **`T`**.

2. When a Java API uses a type variable declared as **`T extends`** ***`Type`***, the clause **`"extends`** ***`Type`*****`"`** is called a **`type`** bound. Use ***`Type`*** in any overridden method instead of **`T`**.

3. When a Java API parameterizes a type, as in ***`PType`*****`<T1, T2, ...>`**, remove the parameterization so it becomes simply ***`PType`***.

The first example above has several instances of these conversions. Consider the Java documentation for ForwardingMap. Notice that it is defined as **`public abstract class ForwardingMap<K, V>`**. Since K and V don't have type bounds, we replace every instance with Object. Although ForwardingMap is parameterized, the BBj class declaration does not include the type parameterization. Also, compare the declarations of **`put()`** and **`putAll()`** below in Java and BBj.

Java

- **`public V put(K key, V value)`**

- **`public void putAll(Map<? extends K,? extends V> map)`**

BBj

- **`METHOD PUBLIC Object put(Object p_key!, Object p_value!)`**

- **`METHOD PUBLIC void putAll(Map p_map!)`**

In many cases, as with this one, the transformation is simple. Just replace **`K`** and **`V`** with **`Object`**, and eliminate the type parameterization on **`Map`**.

XML In A Hurry

On to the grand finale! Consider the sample **`EmployeeSerializer.src`** shown in **Figure 4,** which contains code to take a **`Map`** of the **`Employee`** instances and generate an XML file.

It's all well and good to create an XML document, but what good is an XML document if there is no parser with which to read it? Creating a BBj XML parser would not be impossible, but why go through the effort of implementing and debugging your own parser when standards organizations like the W3 Consortium have implemented every aspect of the XML standard with pain-staking detail and attention?

Of course, now that it's unnecessary to learn another development environment or another programming syntax to use these libraries, the answer is simple: There's no reason to re-code it in BBj when it's already available in Java and can be simply extended. The code in

>>

```
USE java.io.StringWriter
USE java.util.Iterator
USE java.util.Map

USE javax.xml.parsers.DocumentBuilderFactory

USE org.w3c.dom.Document
USE org.w3c.dom.Element

USE org.w3c.dom.ls.DOMImplementationLS
USE org.w3c.dom.ls.LSOutput
USE org.w3c.dom.ls.LSSerializer

USE ::Employee.src::Employee

REM ' Get employees map.
DECLARE Map employeesMap!
employeesMap! = Employee.getEmployees()
fileName$ = "employees.xml"

REM ' Get a DOM Document instance to generate
DECLARE DocumentBuilderFactory factory!
DECLARE Document doc!
factory! = DocumentBuilderFactory.newInstance()
doc! = factory!.newDocumentBuilder().newDocument()

REM ' The root XML element
DECLARE Element employees!
employees! = doc!.createElement("employees")

DECLARE Element empElem!
DECLARE Element attributeElem!
DECLARE Iterator iter!
DECLARE Employee emp!
iter! = employeesMap!.values().iterator()

REM ' For each employee, create an employee element with an id attribute,
REM ' and add child elements for each employee field.
WHILE (iter!.hasNext())
    emp! = CAST(Employee, iter!.next())
    empElem! = doc!.createElement("employee")
    empElem!.setAttribute("id", STR(emp!.getId()))

    attributeElem! = doc!.createElement("first-name")
    attributeElem!.setAttribute("value", emp!.getFirstName())
    empElem!.appendChild(attributeElem!)

    attributeElem! = doc!.createElement("last-name")
    attributeElem!.setAttribute("value", emp!.getLastName())
    empElem!.appendChild(attributeElem!)

    attributeElem! = doc!.createElement("payroll-type")
    attributeElem!.setAttribute("value", STR(emp!.getPayrollType()))
    empElem!.appendChild(attributeElem!)

    attributeElem! = doc!.createElement("position")
    attributeElem!.setAttribute("value", HTA(emp!.getPosition()))
    empElem!.appendChild(attributeElem!)

    REM ' Add the employee element to the root element.
    employees!.appendChild(empElem!)
WEND
```

Figure 4a. Sample code that generates an XML file (*continued in* **Figure 4b.**)

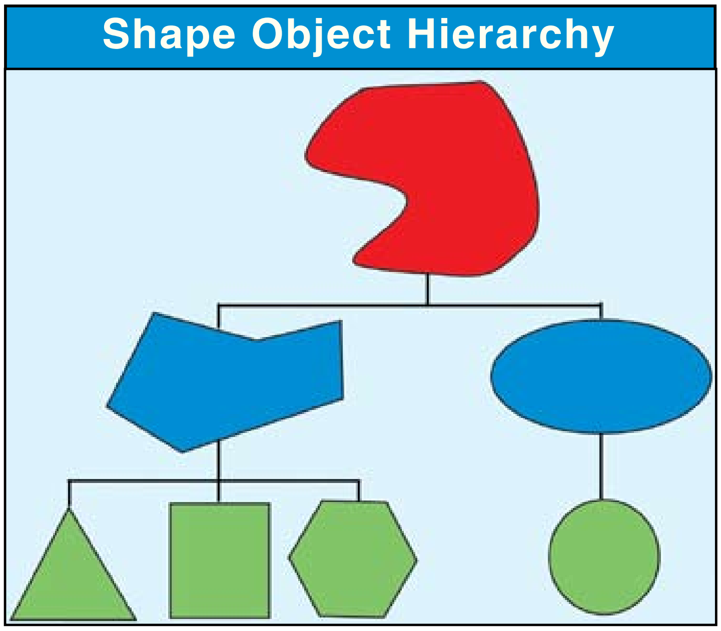
>>

```
REM ' Add the root element to the document
doc!.appendChild(employees!)

DECLARE DOMImplementationLS ls!
DECLARE LSOutput output!
DECLARE StringWriter writer!

ls! = CAST(DOMImplementationLS,
:                   doc!.getImplementation().getFeature("LS", "3.0"))
output! = ls!.createLSOutput()
output!.setEncoding("UTF-8")
writer! = new StringWriter()
output!.setCharacterStream(writer!)

DECLARE LSSerializer serial!
DECLARE BBjString bytes!
serial! = ls!.createLSSerializer()
serial!.getDomConfig().setParameter("format-pretty-print", Boolean.TRUE)
serial!.write(doc!, output!)
bytes! = writer!.toString().getBytes("UTF-8")


REM ' Create backup file if already exists
STRING fileName$,ERR=*NEXT; GOTO BACKUP_DONE
ERASE fileName$ + ".bak",ERR=*NEXT
RENAME fileName$, fileName$ + ".bak"
STRING fileName$

BACKUP_DONE:
ch = UNT
OPEN(ch)fileName$
WRITE RECORD(ch,SIZ=LEN(bytes!))bytes!
CLOSE(ch)

RELEASE
```

Figure 4b. Sample code that generates an XML file

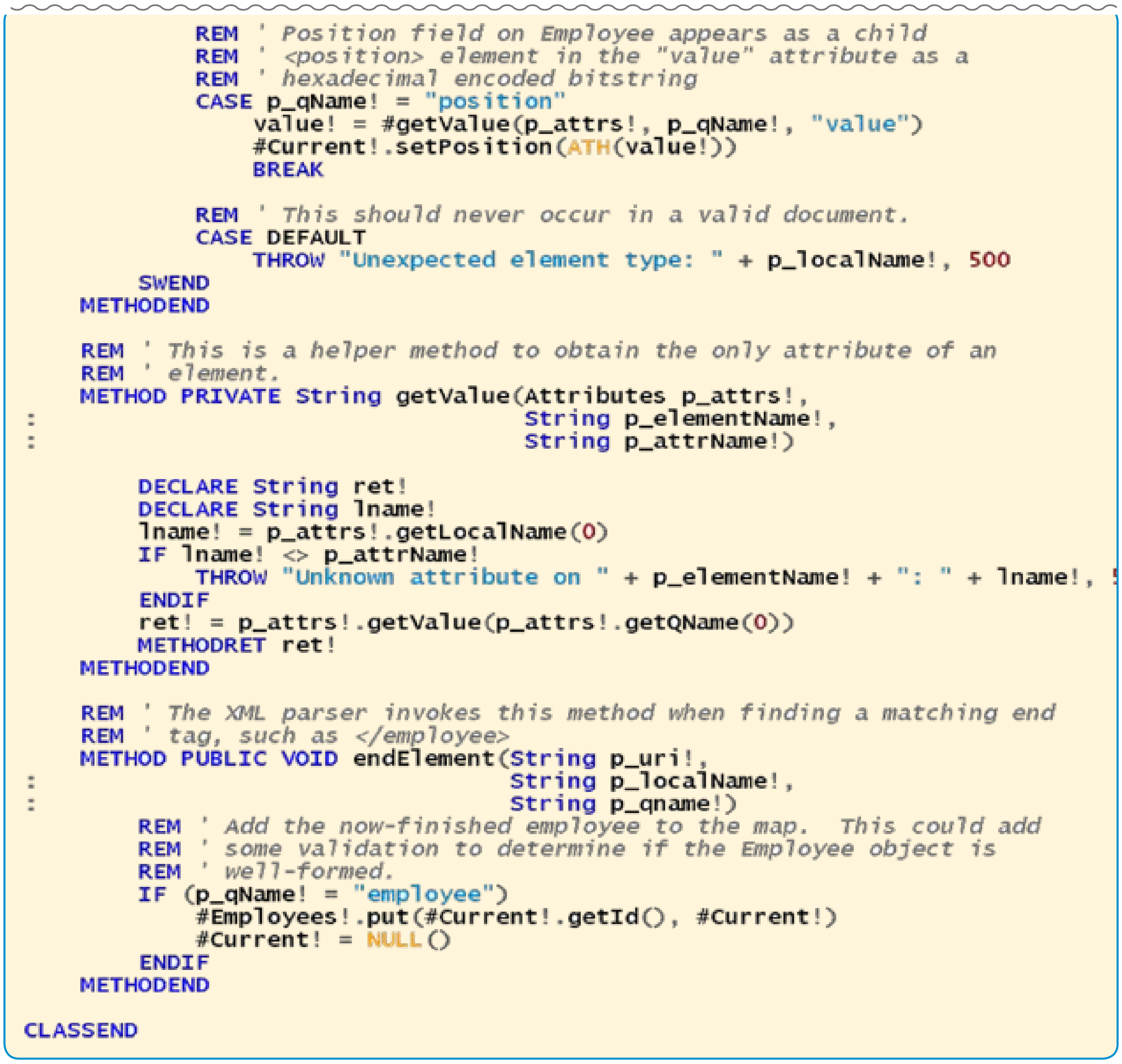
Figure 5 illustrates a parser that reads the XML format written by the previous sample.

The most surprising thing about this sample is what it lacks. Notice the missing **POS()**, **MASK()** and substring notation to perform string manipulation, or **CVS()** functions to strip whitespace. This class simply responds with the appropriate operation when encountering an XML element and attribute names. The sample **EmployeeParser.src** contains all the necessary code to read the included **employees.xml** file. To experiment with other XML syntax, feel free to run **EmployeeParser.src** as a model. >>



```
REM ' EmployeeParser.src

USE java.util.HashMap
USE java.util.Map

USE org.xml.sax.helpers.DefaultHandler
USE org.xml.sax.Attributes

USE ::Employee.src::Employee

CLASS PUBLIC EmployeeParser EXTENDS DefaultHandler
    FIELD PRIVATE Employee Current!
    FIELD PRIVATE Map Employees!

    REM ' The XML parser invokes this method when encountering a start
    REM ' tag in an XML document, such as <employee id="1">.
    REM ' The attributes (id="1") are contained as a map in the given
    REM ' Attributes object.
    METHOD PUBLIC VOID startElement(String p_uri!,
:                                   String p_localName!,
:                                   String p_qName!,
:                                   Attributes p_attrs!)
        DECLARE String value!

        REM ' Perform a particular operation for each tag.  To guarantee
        REM ' document integrity, this might add checks to ensure the
        REM ' elements appear as children of the appropriate parent
        REM ' element.
        SWITCH (1)
            CASE p_qName! = "employees"
                #Employees! = new HashMap()
                BREAK

            REM ' Id field on Employee appears as an attribute of the
            REM ' <employee> tag.
            CASE p_qName! = "employee"
                IF (#Current! <> NULL())
                    THROW "Employee tag inside existing employee tag", 500
                ENDIF
                value! = #getValue(p_attrs!, p_qName!, "id")
                #Current! = new Employee()
                #Current!.setId(NUM(value!))
                BREAK

            REM ' FirstName field on Employee appears as a
            REM ' child <first-name> element in the "value" attribute
            CASE p_qName! = "first-name"
                value! = #getValue(p_attrs!, p_qName!, "value")
                #Current!.setFirstName(value!)
                BREAK

            REM ' LastName field on Employee appears as a
            REM ' child <last-name> element in the "value" attribute
            CASE p_qName! = "last-name"
                value! = #getValue(p_attrs!, p_qName!, "value")
                #Current!.setLastName(value!)
                BREAK

            REM ' PayrollType field on Employee appears as a
            REM ' child <payroll-type> element in the "value" attribute
            CASE p_qName! = "payroll-type"
                value! = #getValue(p_attrs!, p_qName!, "value")
                #Current!.setPayrollType(NUM(value!))
                BREAK

            REM ' Position field on Employee appears as a child
            REM ' <position> element in the "value" attribute as a
            REM ' hexadecimal encoded bitstring
            CASE p_qName! = "position"
                value! = #getValue(p_attrs!, p_qName!, "value")
                #Current!.setPosition(ATH(value!))
                BREAK
```

Figure 5a. A parser that reads XML format *(continued in* **Figure 5b.***)*



```
            REM ' Position field on Employee appears as a child
            REM ' <position> element in the "value" attribute as a
            REM ' hexadecimal encoded bitstring
            CASE p_qName! = "position"
                value! = #getValue(p_attrs!, p_qName!, "value")
                #Current!.setPosition(ATH(value!))
                BREAK

            REM ' This should never occur in a valid document.
            CASE DEFAULT
                THROW "Unexpected element type: " + p_localName!, 500
        SWEND
    METHODEND

    REM ' This is a helper method to obtain the only attribute of an
    REM ' element.
    METHOD PRIVATE String getValue(Attributes p_attrs!,
:                                  String p_elementName!,
:                                  String p_attrName!)

        DECLARE String ret!
        DECLARE String lname!
        lname! = p_attrs!.getLocalName(0)
        IF lname! <> p_attrName!
            THROW "Unknown attribute on " + p_elementName! + ": " + lname!, 5
        ENDIF
        ret! = p_attrs!.getValue(p_attrs!.getQName(0))
        METHODRET ret!
    METHODEND

    REM ' The XML parser invokes this method when finding a matching end
    REM ' tag, such as </employee>
    METHOD PUBLIC VOID endElement(String p_uri!,
:                                 String p_localName!,
:                                 String p_qname!)
        REM ' Add the now-finished employee to the map.  This could add
        REM ' some validation to determine if the Employee object is
        REM ' well-formed.
        IF (p_qName! = "employee")
            #Employees!.put(#Current!.getId(), #Current!)
            #Current! = NULL()
        ENDIF
    METHODEND

CLASSEND
```

Figure 5b. Continuation of a parser that reads XML format

Summary

Many Java libraries exist that do not require any Java programming. But some very high quality libraries accomplish a great deal by providing interfaces or classes that the library designer expects a developer to inherit in order to supply custom behavior. In some cases, libraries define a sequence of operations, but leave the implementation of the operations up to the client code. Other libraries notify the client code of actions that occur in the program, to give the library client a chance to respond to important events. Libraries using these and other similar paradigms were previously less accessible to BBj developers because of the requirement to write Java code. Now, the benefits of these Java libraries are available to all developers simply by writing a Custom Object with the familiar syntax, functionality and support of the BBx® language. ■