Exponentially Better Applications: Embedded Java and BBj™

by David Wallwork

Using BASIS' new product generation, BBj™, programmers will dramatically expand the power, versatility and utility of their applications by embedding Java™ code into their BBj programs. Using embedded code, the BBj application can incorporate any Java class. Java-based tools become BBj tools. But there are some syntax differences: the BBj test code, **`javatest.bbj`**, will look as unfamiliar to the BBx® programmer as it does to the Java programmer. The syntax in **`javatest.bbj`** is a somewhat odd mixture of the syntaxes of the two languages.

BBx programmers will recognize the **`while`** and **`wend`** statements, for example. A Java programmer will understand

```
MyVector = new java.util.Vector();
```

A BBj programmer will understand it all.

This test application, **`javatest.bbj`**, reads from a file, creates a **`java.lang.Vector`** that contains lines from the file, then prints the Vector. To use this code, of course, requires that one have BBj installed.

There are two things about **`javatest.bbj`** that are different from a traditional BBx program:

- The new keywords **`SET_CASE_SENSITIVE_ON`** and **`SET_CASE_SENSITIVE_OFF`**
- The ability to use Java classes and Java syntax while case-sensitive is on

Embedding Java code within a BBj program is really quite easy, although there are some constraints. The following sections give a rather complete survey of those constraints.

Javatest.bbj

```
REM create a file from which we can read data

Call "writefile.bbj"

SET_CASE_SENSITIVE_ON

REM here comes the java

MyVector = new java.util.Vector();

WHILE(1)

    READ(2,END=DoneReading)A$

    myVector.addElement(A$)

WEND


DoneReading:
z = myVector.size();
FOR I = 0 TO z-1
```

```
    PRINT myVector.elementAt(Integer.parseInt(A$))
NEXT I
SET_CASE_SENSITIVE_OFF
```

Rem note that we can not access the variable z now that case-sensitive

```
Rem is off. When we attempt to print z, we instead print a new
variable

Rem whose name is Z which is a different variable from z. Had we
named

Rem our variable Z within the region that was case-sensitive, we
WOULD

Rem be able to access it here.
```

Print z
Print Z

Writefile.bbj

```
Open(2,err =
nofile)"myTest.txt"
Goto fileExists
nofile: string
"myTest.txt"
open(3)"myTest.txt"
for i = 1 to 10
    print(3) "This is
line", I
next I
close(3)
retry
fileExists:
exit
```

This TechCon99 session will discuss Java™ terms and concepts with which some BBx developers may be unfamiliar. The following Web sites have more information and tutorials on the issues that will be presented in this article and in this session.

http://java.sun.com/docs/faqindex.html
http://devdentral.iftech.com/learning/tutorials/java/javaintro/

http://metalab.unc.edu/javafaq/javatutorial.html

Specific Differences

There are three new capabilities in BBj that give you access to all the functionality of the Java language.

1. You can now instantiate Java objects within BBj code. When the BBj Interpreter encounters the code

   ```
   myInteger = new java.lang.Integer()
   myString = new java.lang.String("123")
   ```

   it will instantiate a new Integer and a new String and will store them in its collection of variables with the associated names of `myInteger` and `myString`.

2. You can access the public attributes of a variable that contains a Java object. When the BBj Interpreter encounters the code

   ```
   I = myInteger.MAX_VALUE
   ```

   it will assign the value to a variable with the name `I`.

3. You can access the public methods of a variable that contains a Java object. When the BBj Interpreter encounters the code

   ```
   B$ = myInteger.toString()
   J = Integer.parseInt(B$)
   ```

   it will assign appropriate values to **`B$`** and to **`J`**.

Maintaining Compatibility

One of the primary requirements of BBj is to maintain compatibility between BBx applications and BBj applications. Therefore, any conflicts between the syntax or paradigms of BBx and Java are resolved in favor of the BBx language. This results in some syntax differences between BBj and Java. Namely:

- BBj does not enforce type-checking at parse-time

  In BBx (and hence in BBj) a statement stands alone. It is validated (and can be executed) as a unit unto itself. Consider the Java code:

  ```
  String myString;
  Integer myInteger;
  myString = new String();
  myInteger = new Integer();
  myInteger = myString;
  ```

  This code will not compile in a Java compiler because Java will not allow the assignment of a string to a variable that has type Integer.

  In BBj, this code is acceptable. Since the line

  ```
  MyInteger = myString
  ```

  must stand alone as a complete unit, BBj will take whatever is in the variable named **`myString`** and associate it with the variable name **`myInteger`**.

- BBj does not enforce name-checking at parse time

  When the BBj Interpreter encounters the line

  ```
  X = myInteger.MAX_VALUE
  ```

  it interprets that line as a complete unit. It will attempt to access the attribute **`MAX_VALUE`** in whatever object is associated with the name **`myInteger.`** Because the Java compiler can enforce strict type-checking, it will know at compile time whether the variable **`myInteger`** has an attribute **`MAX_VALUE`** and will not compile this line if it does not.

- Import Statements are not available in BBj

  The syntax of import statements conflicts with that of TemplatedStrings and so are not available in BBj to maintain compatibility with BBx.

- Try, catch constructs are not supported in BBj although BBj may call a method of a Java class that contains a catch/throw construct.

- BBj programmers will need to be aware of the status of `'SET_CASE_SENSITIVE_ON'` and `'SET_CASE_SENSITIVE_OFF'`.

  When case-sensitive is off (the normal BBx behavior), the line

  `'x = 5'` will create a new variable. The name of the variable will be promoted to uppercase. So the program will have a variable whose name is `X` (uppercase) and whose value is 5. If case-sensitive is on, the line `'x = 6'` will create a second variable whose name is `x` (lowercase) and whose value is 6. While case-sensitive is on, the statement `'PRINT X'` and `'PRINT x'` will print `'5'` and `'6'`, respectively. When case-sensitive is off, each of these statements will refer to the variable `X` (uppercase) and so they will each print `'5'`.

  The keywords (e.g. `PRINT,REM,WHILE`) must be in uppercase. If case-sensitive is off, the parser will promote them automatically, but if case-sensitive is on, it will not. So the statement **'print 123'** will execute as expected when case-sensitive is off but will result in a syntax error when case-sensitive is on.

  Like variable names and keywords, label names are case sensitive when case-sensitive is on. So in `javatest.bbj`, the statement `'GOTO DoneREading'` would not have been correct.

- BBj does not recognize Java control statements. This means, for example, that the BBj programmer must use `WHILE/ WEND` rather than the Java `while() { }` construct.

Java Type-Casting

Java has strong type-casting that is enforced by the requirement to declare the type of every variable. BBx has weak type-casting that is enforced by the naming convention that `A` is a number while `A$` is a string and `A%` is an integer. BBj will accept code such as the following

```
A$ = "hello"
B = 55
B = A$
```

BBj programmers will learn that type-cast errors will pass the parsing and will result in run-time errors.

To avoid this, it is recommended that the programmer separate Java code from non-Java code as much as possible. If you want to include an extended block of Java code, then place that code in a 'helper' class. Compile the helper class using your Java compiler and then call the methods of your helper class from within BBj rather than including your entire code block within BBj. In this way, you will get the benefits of type-checking your code block but will still be able to execute your code block from within BBj.

Using Java Sockets

This last example gives a sense of the power of embedded Java in BBj. With just a few lines of Java code, sockets are established between two BBj applications. Type something in the console of one application and it appears in the console of the other. All of the work required to create a socket connection is handled transparently by Java objects embedded in the programs by the BBj developer, allowing the developer to concentrate on adding unique value to the application.

SocketServer.bbj

```
SET_CASE_SENSITIVE_ON
  port$ = "1122"
  serverSocket = new java.net.ServerSocket(Integer.parseInt(port$));

  PRINT "Listening for connections on port "


 Socket = serverSocket.accept();
  PRINT "Connection established"
 InputStream =
    new java.io.ObjectInputStream(socket.getInputStream());
  FOR N = 1 TO 10


    A$ = inputStream.readObject()
    PRINT A$
  NEXT N

SET_CASE_SENSITIVE_OFF
```

SocketClient.bbj

```
SET_CASE_SENSITIVE_ON

 port$ = "1122"
  clientSocket = new java.net.Socket("localhost",Integer.parseInt(port$));
  PRINT "Client connected "
  PRINT ""
  PRINT ""
  PRINT "Please enter message to be sent to client"
  OutputStream =
    new java.io.ObjectOutputStream(clientSocket.getOutputStream());
  FOR N = 1 TO 10
    READ A$
    outputStream.writeObject(A$)
    NEXT N

SET_CASE_SENSITIVE_OFF
```

David Wallwork joined BASIS in March 1998 as a Software Architect. He has been the technical lead on the design and development of BBjÊ . David has a Master's degree in Mathematics from the University of California at Berkeley and a Master's degree in Computer Engineering from the University of New Mexico.