

Object-Oriented Syntax and Runtime Enhancements in BBJ 9.00

April 2009

By

Adam Hawthorne

Introduction
Inner Classes
Java Interfaces
Performance Enhancements
Types and Conversions
Automatic USE from java.lang

Introduction [\(back to top\)](#)

Since the initial release of BBJ®, developers have had the opportunity to use Java objects in their BBx® programs. This significantly expands the number of libraries that are available to the BBJ developer, giving opportunities to leverage the very active Java community. Not only can BBJ programs be run on any supported Java platform, but the wide array of tools and technologies associated with the Java platform have been available in BBJ from the very beginning.

In BBJ 6.00, BASIS continued the commitment to provide features that allow developers to make use of current technology trends. By providing the ability to develop programs using modern object-oriented programming (OOP) techniques through the introduction of CustomObjects, BASIS provides BBJ developers access to the tools available in Java as well as the ability to use programming practices that are prevalent in many modern programming languages. BBJ 8.00 expanded the use of embedded Java objects on the server to allow client-side Objects through the use of [ClientObjects](#).

BBj 9.00 continues to introduce new features and improvements to these core aspects of the BBx language as introduced below.

Performance Enhancements [\(back to top\)](#)

BBj 9.00 provides significant performance improvements to CustomObject method calls. All method calls are now faster than in previous releases. Different types of method calls saw different improvements, but the most modest improvements are still 30% faster in BBJ 9.00. In the best case, the time to perform a particular method call decreased by a factor of 134 when all the methods exist in the same program. The performance improvements are even more noticeable when the CustomObject definitions span several files. Notably, the time to invoke a method is now virtually the same as the time to perform a CALL when both invocations have the same number of parameters. Even the venerable CALL verb saw a small boost in performance, about 4%, in BBJ 9.00.

There are even greater benefits by leveraging some of the existing language tools. BBJ 6.00 introduced the DECLARE verb to assign a type to a particular variable. The DECLARE verb instructs BBJ to enforce static type checking rules on expressions involving variables used in a DECLARE statement. In that release, DECLARE allowed for BBJCpl to warn about type-checking errors via the `-t` command line option, and for the BASIS IDE to provide code completion on variables with DECLARE statements.

BBj 9.00 further improves the benefit of using DECLARE statements. Method calls that only use variables specified by DECLARE statements (or expressions on those variables) can reach 90% improvement in execution time. Even the simplest method invocations are up to 10% faster than the same method call that includes an expression with an undeclared variable. DECLARE statements allow BBJ to infer guarantees about the object code in BBJ programs that may lead to even further enhancements in the future.

Java Interfaces [\(back to top\)](#)

BBj 9.00 also adds another feature to allow developers to more easily make use of existing Java libraries without having to write any Java code: The ability to implement a server-side Java interface on a CustomObject class.

To implement a Java interface, simply add the name of the Java interface to the IMPLEMENTS clause of a CustomObject class definition. An example using the `java.lang.Comparable` interface:

```
REM ' Comparable is in java.lang
CLASS PUBLIC DateOrderNumKey IMPLEMENTS Comparable
  FIELD PRIVATE BBJNumber Date
  FIELD PRIVATE BBJNumber OrderNum

  REM ' Constructor takes the date and the order number to use in
  REM ' the compareTo() method.
  METHOD PUBLIC DateOrderNumKey(BBJNumber p_date, BBJNumber p_orderNum)
    #Date = p_date
    #OrderNum = p_orderNum
  METHODEND

  REM ' This method is specified by the java.lang.Comparable
  REM ' interface. Notice it returns 'int' and takes 'Object' .
  REM ' It should return an integer less than zero if this object
  REM ' should sort less than p_other!, 0 if this object sorts the
  REM ' same as p_other!, and an integer greater than zero if this
  REM ' object should sort greater than p_other! .

  METHOD PUBLIC int compareTo(Object p_other!)
    REM ' Get the other DateOrderNumKey .
    DECLARE DateOrderNumKey other!
    DECLARE BBJNumber val

    other! = CAST(DateOrderNumKey, p_other!)

    REM ' First sort by date by seeing which date is bigger.
    val = #Date - other!.getDate()

    REM ' If the date values are equal, then we look at order_num
    IF (val = 0) THEN val = #OrderNum - other!.getOrderNum() FI

    REM ' Finally, we return val
    METHODRET val
  METHODEND
CLASSEND
```

See `CustomKey.src` for a complete sample.

```
REM ' Java classes we'll use
USE java.util.TreeMap
USE java.util.Iterator

REM ' This class implements a DateOrderNumKey that sorts first by date
REM ' And then by "order number"

REM ' Comparable is in java.lang
```

```

CLASS PUBLIC DateOrderNumKey IMPLEMENTS Comparable
  FIELD PRIVATE BBJNumber Date
  FIELD PRIVATE BBJNumber OrderNum

  REM ' Constructor takes the date and the order number to use in
  REM ' the compareTo() method.
  METHOD PUBLIC DateOrderNumKey(BBJNumber p_date, BBJNumber p_orderNum)
    #Date = p_date
    #OrderNum = p_orderNum
  METHODEND

  REM ' This method is specified by the java.lang.Comparable interface.
  REM ' Notice it returns int and takes Object. It should return an
  REM ' integer less than zero if this object sorts less than p_other!,
  REM ' 0 if this object sorts the same as p_other!, and an integer
  REM ' greater than zero if this object sorts greater than p_other!.
  METHOD PUBLIC int compareTo(Object p_other!)
    REM ' Get the other DateOrderNumKey.
    DECLARE DateOrderNumKey other!
    DECLARE BBJNumber val

    other! = CAST(DateOrderNumKey, p_other!)

    REM ' First sort by date by seeing which date is bigger.
    val = #Date - other!.getDate()

    REM ' If the date values are equal, then we look at order_num
    IF (val = 0) THEN val = #OrderNum - other!.getOrderNum() FI

    REM ' Finally, we return val
    METHODRET val
  METHODEND
CLASSEND

REM ' Set up our template and our data file, if necessary.
template$ = "order_num:I(4),order_date:C(8),desc:C(20*=0)"
DIM data$:template$
ch = UNT
OPEN (ch,ERR=CREATE_FILE)"orders.mkv"

REM ' Create the TreeMap we'll use to order our keys
DECLARE TreeMap map!
map! = new TreeMap()

REM ' Read through our data file and use our implementation of Comparable
REM ' to insert into the TreeMap
key$ = KEYF(ch)
DECLARE DateOrderNumKey key!
WHILE 1
  READ RECORD(ch,KEY=key$)data$
  key! = new DateOrderNumKey(JUL(data.order_date$), data.order_num)
  map!.put(key!, data$)
  key$ = KEY(ch,END=*BREAK)
WEND

REM ' Get an iterator for the map.
DECLARE Iterator iter!
iter! = map!.values().iterator()

```

```

REM ' The iterator will iterate in the order defined by our Comparable,
REM ' which sorts first by date and then by order_num
date$ = ""
WHILE (iter!.hasNext())
    data$ = CAST(BBjString, iter!.next())
    IF (date$ <> data.order_date$) THEN PRINT data.order_date$ FI
    date$ = data.order_date$
    PRINT "          ", data.order_num:"00000 ", data.desc$
WEND

STOP

CREATE_FILE:

REM ' Populate the data file with some dummy data
MKEYED "orders.mkty",[1:4],0,40
ch = unt
OPEN (ch)"orders.mkty"
data.order_num = 1
data.order_date$ = DATE(JUL(2009,4,1))
data.desc$ = "Order 1"
WRITE RECORD(ch)data$

data.order_num = 2
data.order_date$ = DATE(JUL(2009,4,2))
data.desc$ = "Order 2"
WRITE RECORD(ch)data$

data.order_num = 3
data.order_date$ = DATE(JUL(2009,4,2))
data.desc$ = "Order 3"
WRITE RECORD(ch)data$

data.order_num = 5
data.order_date$ = DATE(JUL(2009,4,2))
data.desc$ = "Order 5"
WRITE RECORD(ch)data$

data.order_num = 4
data.order_date$ = DATE(JUL(2009,4,3))
data.desc$ = "Order 4"
WRITE RECORD(ch)data$
CLOSE(ch)

RETRY

```

Another example in the same vein is to sort a list of strings according to their numeric value instead of their lexicographic ordering. This uses the `java.util.Comparator` interface:

```

USE java.util.Comparator
USE java.util.Collections

CLASS PUBLIC NumericStringSorter IMPLEMENTS Comparator

    METHOD PUBLIC int compare(Object p_o1!, Object p_o2!)
        DECLARE String s1!
        DECLARE String s2!

```

```

    s1! = CAST(String, p_o1!), s2! = CAST(String, p_o2!)
    n1 = NUM(s1!, ERR=*NEXT); val1 = 1
    n2 = NUM(s2!, ERR=*NEXT); val2 = 1

    IF (val1 AND val2) THEN METHODRET n1 - n2 FI

    IF (val1) THEN METHODRET -1 FI
    IF (val2) THEN METHODRET 1 FI

    METHODRET s1!.compareTo(s2!)
METHODEND
CLASSEND

REM ' Our Comparator
DECLARE Comparator sorter!
sorter! = new NumericStringSorter()

DECLARE BBjVector v0!
DECLARE BBjVector v1!
DECLARE BBjVector v2!
v0! = BBJAPI().makeVector()

v0!.addItem("100")
v0!.addItem("1")
v0!.addItem("22")
v0!.addItem("3")
v0!.addItem("4402")
v0!.addItem("Joe")

v1! = CAST(BBjVector, v0!.clone())
v2! = CAST(BBjVector, v0!.clone())
REM ' Sort lexicographically (default for String)
Collections.sort(v1!)

REM ' Sort using our Comparator
Collections.sort(v2!, sorter!)

REM ' Print the results
PRINT v0!.toString()
PRINT v1!.toString()
PRINT v2!.toString()

```

Besides these examples of sorting, there are many third party libraries that define interfaces for a programmer to implement. Some examples include:

Google Collections API: <http://code.google.com/p/google-collections/>

The Apache Commons libraries: <http://commons.apache.org/>

Jasper Reports: http://jasperforge.org/website/jasperreportswebsite/trunk/index.html?group_id=252

Inner Classes [\(back to top\)](#)

BBj 9.00 now permits references to "inner classes" in program code. Inner classes in Java are used to group tightly-coupled classes by including them in the same class as the class that uses them. Often, they serve as enumeration constants or configuration parameters. An instance in which this has made programming difficult in the past is in using a `java.util.Map`, where the `Map.entrySet()` method returns a `java.util.Set` containing the `Map.Entry` pairs

defined in the current Map. It was previously impossible to declare a variable of type `Map.Entry` to avoid warnings in the type checking option of `bbjcl`.

It is also possible to invoke a static method or to call a constructor of an inner class. These were previously impossible without using the Reflection APIs. Many of the classes `java.awt.geom` package have inner classes `Double` and `Float` that define whether the shape is composed of Java `float` or `double` values. Refer to the recent Advantage magazine article [Inner Types](#) and [downloadable samples](#) that illustrate this feature.

Types and Conversions [\(back to top\)](#)

BBj 9.00 allows use of the Java primitive types in type references, as can be seen from some of the earlier samples that refer to `int`. These are allowed in `DECLARE` statements, method parameters and return types, and in the `CAST()` function.

It also improves semantics for conversions between `byte[]` and `java.lang.String` in Java methods, including well-defined conversions to and from the legacy BBj string type. The conversion rules now allow any conversion from `BBjString` to `byte[]` or `String`. If these defaults do not fit a particular use case, simply use the `CAST()` function to force a particular conversion. The following table defines the rules:

Expression Type	<code>foo(byte[])</code>	<code>foo(String)</code>	<code>foo(Cloneable)</code> OR <code>foo(Object)</code>	<code>foo(String)</code> AND <code>foo(byte[])</code>	<code>foo(Object)</code> AND <code>foo(byte[])</code>
<code>BBjString</code>	Copies <code>byte[]</code> , passes <code>byte[]</code> to method.	Uses the platform encoding to generate a <code>String</code> , passes to the method	Calls method with a <code>String</code> argument as in <code>foo(String)</code>	Calls <code>foo(String)</code>	Calls <code>foo(byte[])</code>
<code>String</code>	Uses platform encoding to generate a <code>byte[]</code> , passes to the method.	Calls method	Calls method	Calls <code>foo(String)</code>	Calls <code>foo(Object)</code>
<code>byte[]</code>	Calls method	Uses the platform encoding to generate a <code>String</code> , passes to the method	Calls method	Calls <code>foo(byte[])</code>	Calls <code>foo(byte[])</code>

In addition, the `BBjNumber` conversions are now explicitly built into the type system, which allows the typechecker to catch more potential runtime errors. For example, code which passes a literal number with a non-zero fractional part to a `BBx` function taking an integer argument or to a method taking a Java `int` will cause an error in `bbjcl` when using the type checking option.

As hinted at by the `BBjString` conversions, the `CAST()` function now performs value conversions for the Java numeric types. Casting a result of type `float` to an `int` will truncate the fractional part, and expressions that produce `BBjNumber` can be cast to a `float` or `double` as needed.

The cryptic expression `new java.lang.Float(value).floatValue()` can be replaced with the much more obvious `CAST(float, value)`. Also, expressions of type `java.math.BigDecimal` may be converted to and from a `BBjNumber`.

Automatic USE from java.lang [\(back to top\)](#)

Although small, this feature improves consistency when concurrently programming in both Java and BBj. When programming Java code, the Java compiler automatically imports types from the `java.lang` package such as `java.lang.Object` and `java.lang.String`. With the BBj 9.00 release, BBj also automatically imports all classes from `java.lang`. These automatic imports are the very last types searched, and will not change the behavior of existing programs. There is one caveat to the automatic importing, although it is unlikely to introduce any serious inconvenience: the type `java.lang.Void` is hidden from the programmer by virtue of the case-insensitivity of the `VOID` type keyword in BBj.

These enhancements continue to streamline development for new software, provide simplified access to third-party libraries and increase performance. Developers who may have shied away the prospect at setting up a Java development environment can now dabble in Java development directly from BBj programs using interfaces. Developers who use Java regularly can take advantage of the ability to call interface methods on `CustomObjects` in their Java code directly and can implement code with `CustomObjects` without being concerned about performance. And everyone can appreciate the smaller improvements that make developing in BBj even easier.

