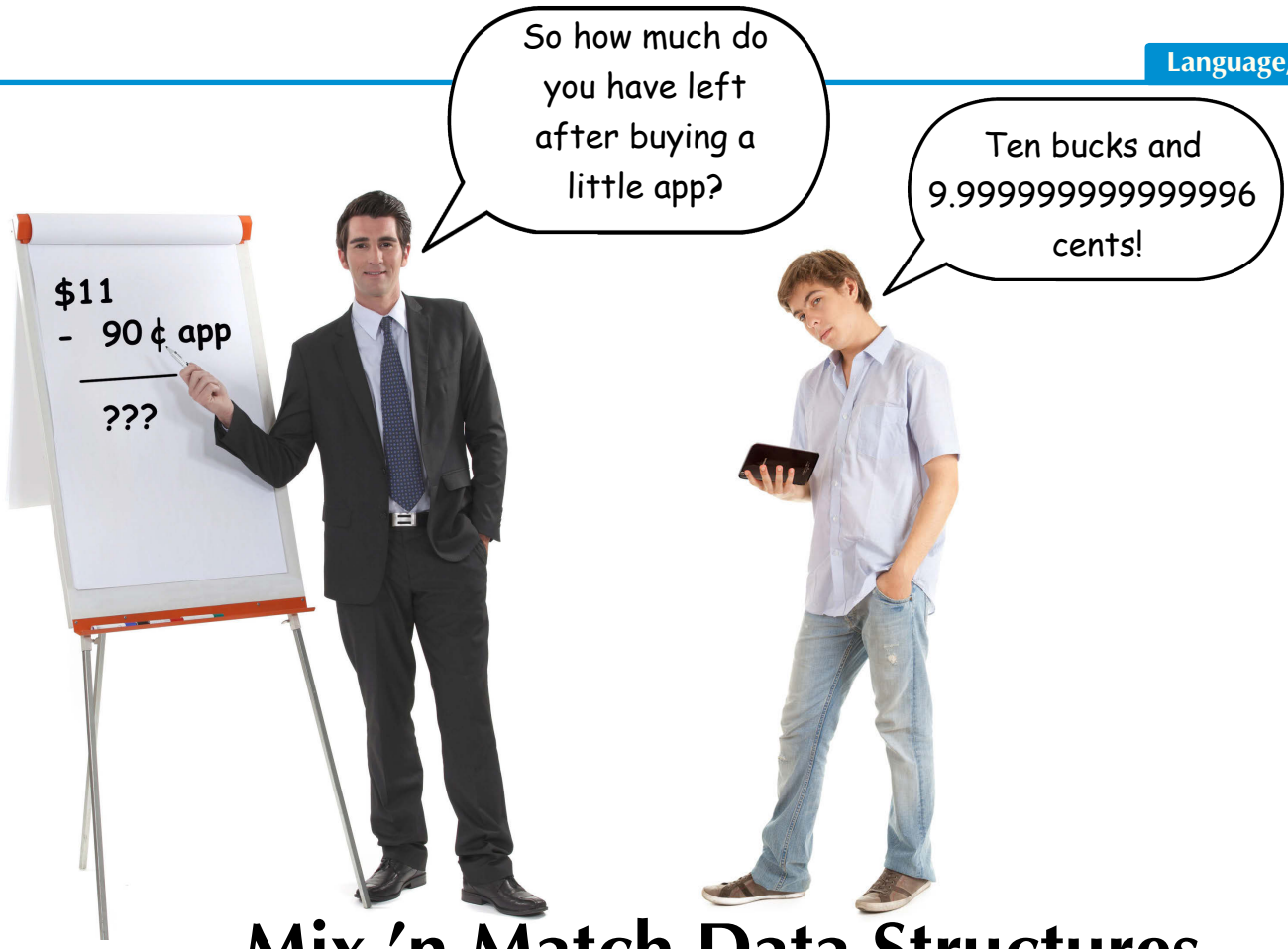

Mix 'n Match Data Structures Between BBj and Java

One of the key advantages of BBj® is its seamless integration with the Java platform on which it runs. Developers can use Java objects within their BBx® applications as though they were native BBj objects. But on the rare occasion, a developer must use some of the more advanced features of BBj to deal with a semantic mismatch one may encounter between BBx, a language with a 30-year lineage, and Java, having a far briefer teenage lineage. This article discusses some of the tips and tricks as well as some recent improvements to the BBj version of the BBx language and in the BBj API that provide help in these situations.

Character Encoding

When two countries with different languages share a border, there is always an area where the two cultures clash. The issues surrounding character encoding between BBx and Java is similar at multiple levels.

By Adam Hawthorne
Software Engineer

The term "character encoding" would be better called "character transcoding." A character set (or "charset") defines a mapping between a range of integers and a set of symbols that each represents an atomic unit of written language, known as a "character." A character encoding defines the algorithm by which the integers from a given charset are then encoded as a sequence of 8-bit bytes, and also by which a sequence of bytes are then decoded back into valid integers in that charset.

The engineers who created Java chose to use the Unicode character set as their mapping between integers and symbols. Unicode strives to assign a number to every distinct character. Internally, Java must represent the integer associated with each character as a sequence of bytes. Java does this by representing each integer as one or more 16-bit characters in a character encoding known as UTF-16. These characters are held in an object called a **`java.lang.String`**, or a "Java String."

BASIS created BBx before Unicode was even a twinkle in its creators' eyes, in a simpler time when 64KB really was enough for anyone. The idea of using two bytes for every character would have been an unthinkable waste of space, and the state of the art was to use an encoding in which, at most, 256 character symbols could each be represented by a single byte. This relationship between bytes and characters became an assumption on which much of the language and tools depends even today. The familiar string variable X$, literal string constants such as **`"abcdefg`**," and hexadecimal string constants such as **`$61626364656667$`** are all stored internally as a sequence of 1-byte values.

The boundary between Java and BBx where a BBx string of bytes is converted into a Java String is hidden for the most part by using the character encoding defined by the operating system. On UNIX systems, Java uses the contents of the **LC_ALL, LC_CTYPE** or LANG environment variables to determine the default character encoding. On Windows systems, Java uses the user's Control Panel preferences.

The Java notion of a capital "C" **Charset** includes an encoding and decoding algorithm to convert from a sequence of bytes that represent the integers corresponding to the characters from that charset into a sequence of 16-bit integers that represent those same characters in Unicode (and back).

When passing a byte-per-character BBx string value into a Java method that takes a Java String, BBj uses the default **Charset** to convert from those bytes into a Java String. When returning a Java String from a Java method, BBj uses the default **Charset** to convert from the String back into bytes.

This implicit algorithm has the potential to introduce unintended behavior. However, combined with the Java API, BBjAPI contains several methods to defeat or modify the implicit behavior by which we can smooth over these differences between the Java language and the BBx language. See a comparison of the different methods in **Figure 1**.

METHOD	DESCRIPTION	USE/RATIONALE
`BBjAPI::asBytes(string)*`	Obtain an instance of the `Java byte[]`object that represents a BBx string.	There may be two methods with the same name; one that takes a `java.lang.String` and one that takes a `byte[]`. Use the result of this method as an argument to select the one that takes a `byte[]`, otherwise BBj will use the method that takes a `java.lang.String`. BBj will automatically convert any BBx string value passed to a method that takes a `java.lang.Object` to a `java.lang.String`. This conversion can result in a loss of data, especially if the data is not textual in nature. Use the result of this method to avoid any automatic conversion for a `java.lang.Object` parameter type.
`BBjAPI::toLocal(string)`	Obtain the unmodified encoding for characters obtained from a `java.lang.String`.	BBj converts bytes that do not map to a valid character in a particular charset to a special range of characters in Unicode called the "Private Use Area". These characters appear as UTF-16 values `0xE0FF-0xE1FF`. Using this method prevents a `String` containing characters in this range from being translated back into potentially valid byte values. This might be important if the data in the String represents something other than character data and should not be modified in any way. Each encoding has a unique byte sequence to represent a character that does not have a valid encoding. For Cp1252, this is the byte value `0x3F`, which corresponds to the question mark character `'?'`.
`BBjAPI::toUnicode(string)`	Obtain a pure Unicode **`java.lang.String` for a `byte[]`** or BBx string.	BBj converts bytes that do not map to a valid character in a particular charset to a special range of characters in Unicode called the "Private Use Area". These characters appear as UTF-16 values `0xE0FF-0xE1FF`. Using this method prevents this translation from occurring, and any invalid bytes will be replaced with the replacement character value `0xFFFD`. This could be useful for determining that a certain sequence of bytes is invalid character data.
`new String(byte[], String)`	Create a `java.lang.String` interpreting the byte[] argument with a specified encoding named by the `String` parameter.	Use this if you are receiving `byte[]` data from another source and the `byte[]` might have been generated using a different charset than the platform charset. E.g., consider a situation where a Windows computer generates byte data using the Cp1252 charset, and inserts that data into a data file. Then, the data must be used on a Linux server using the ISO-8859-15 charset. Use this method to transform that data into a `java.lang.String: str! = new String(x$, "Cp1252")`. Then, use `toLocal(str!)` to transform the `java.lang.String` back into a BBx string.
`String.getBytes(String)`	Create a `byte[]` or BBx string value using a particular encoding named by the `String` parameter	This method creates a sequence of bytes using a specified character encoding. For instance, assume one must create a document for a partner using the UTF-8 character encoding. Once the contents document exist in a string `x$`, one might use the following code to obtain the UTF-8 bytes: `y$ = toUnicode(x$).getBytes("UTF-8").` Then, write the string `y$` to a file.

Figure 1. Summary of the different methods and their uses

Numbers and "Business Math"

One of the original motivations for creating Business BASIC was to address the sometimes-counterintuitive behavior of traditional binary floating point numbers. Binary floating-point data types favor speed and a compact representation using base 2 arithmetic. Numbers using so-called "business math," also known as "binary coded decimal" (BCD), favor predictable, intuitive results using base 10 arithmetic that match calculations made by hand. Since BBx uses BCD operations for real number calculations, traditional BBx programs avoid the confusion that arises from the behavior of binary floating-point calculations.

The Java programming language, on the other hand, derives much of its syntax and semantics from the "C" family of languages, which exposes the native binary floating-point operations of the underlying hardware via its primitive **float** and **double** data types. Now that BBj provides access to the massive Java ecosystem, the differing behaviors of the traditional and business approach to real numbers may surprise BBx developers when using libraries that expect Java **double** or **float** types.

BBj once again smooths over many these differences by automatically converting from its internal BCD representation to the binary floating point representation. However, there are two inconsistencies BBx developers should be aware of:

1. The Java **float** and **double** types cannot store exact values for many common decimal values. When converting from one of these floating point types into a BBx numeric type, you may wish to pass it to the ROUND() function first. Consider the output from this program:

```
PRECISION 16
a = new Double("10.1")
PRINT a
a = ROUND(new Double("10.1"), 2)
PRINT a
```

Output:

```
10.0999999999999996
10.1
```

The value 10.1 cannot be represented as a finite sequence of numbers in base 2. Neglecting to take that into consideration can produce confusing results.

2. Java does not allow automatic conversions (called "narrowing conversions") from a **double** to a **float**. BBj strives for consistency with Java by disallowing an automatic conversion from a BBx numeric type directly to a **float** However, the CAST() function allows a program to perform this conversion:

```
DECLARE float a!
a! = CAST(float, 10.1)
print a!
```

Output:

```
10.1000003814697266
```

For in-depth information describing the gory details of binary floating point representation, visit tinyurl.com/czqwkr9.

Java Arrays

With more data comes more ways to interpret that data, even with the most simple of compound data types: the array. BBx implements an array type as a contiguous block of memory with metadata that provides the information about the dimensional structure of the array. Java arrays are implemented as nested "arrays of arrays," where each element in any intermediate arrays may refer to an individual array. See both storage layouts illustrated in **Figure 2**.

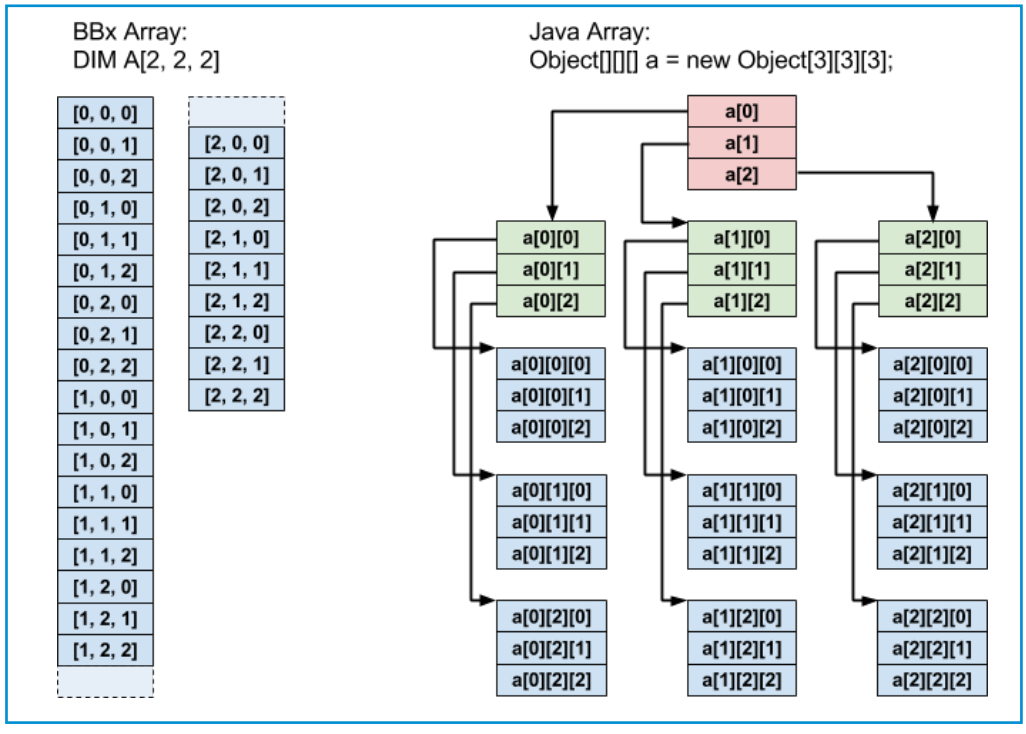

Figure 2. BBx and Java array examples

Because of this mismatch, BBj converts from Java arrays to BBx arrays to help when interacting with Java. A method or object variable that returns a Java array may be assigned to a BBx object array variable, e.g.:

```
locales![] = java.util.Locale.getAvailableLocales()
```

The Java array is treated as a 1-dimensional array, regardless of the number of dimensions used to create it originally. The elements of the BBx array may contain other Java array values.

BBjVector

BBj also provides the **BBjVector** class to bridge the gap between Java arrays and BBx syntax. Use BBJAPI().makeVector() to obtain an instance of a BBjVector. BBjVector can be passed to any method that receives a Java array parameter, as long as the individual elements in the **BBjVector** are all assignable to the component type of the array. For example, one may use the **String.format(String, Object[])** method to create a formatted string from Java. The following program:

```
vec! = bbjapi().makeVector()
vec!.addItem(new Double("68.2"))
vec!.addItem("Fahrenheit")
print String.format("The temperature is %.1f degrees %s.", vec!)
```

produces the result:

```
The temperature is 68.2 degrees Fahrenheit.
```

The parameter **vec!** is automatically converted into an **Object[]**.

Java Class and Package Names

Java packages, classes, methods, and fields may all begin with the underscore '_'. It is now possible to refer to these Java productions from a BBj program. Here is the motivating example, supported in BBj 13.0 and above:

```
USE com.microsoft.schemas.exchange.services._2006.messages.ExchangeService
```

This feature also enables the use of an underscore at the beginning of a standard variable name:

```
_s$ = "Hello, world!"
_n = 10
_o! = new Object()
```

Summary

BBj does most of the heavy lifting to hide the inconsistencies between BBx and Java. Most BBx applications never need to concern themselves with the level of detail described here. But if these differences ever do cause your application or your data to misbehave, this view behind the curtain can give you the tools you need to put it back in line. ■ links.basis.com/13code