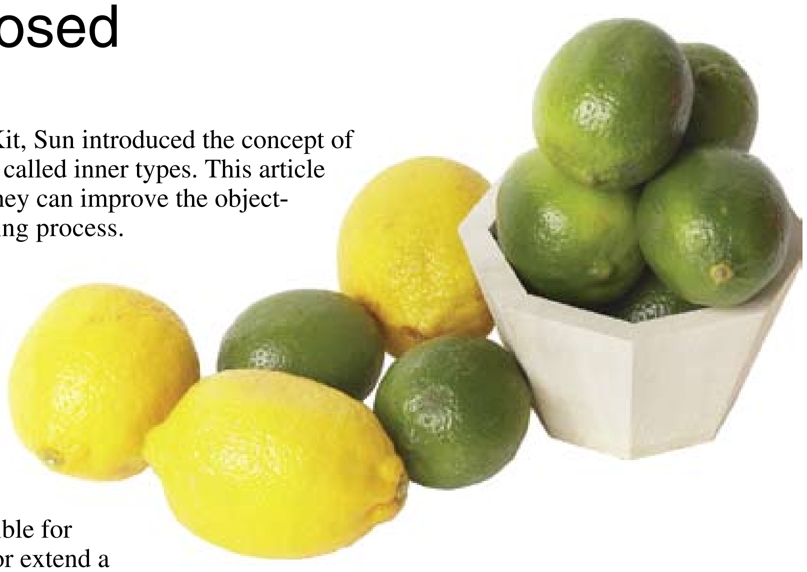
Inner Types Exposed

By Adam Hawthorne



In version 1.1 of the Java Development Kit, Sun introduced the concept of inner classes and interfaces, collectively called inner types. This article explains what inner types are and how they can improve the object-oriented BASIS developers' programming process.

What They Do

Inner types, or types defined within another type, provide the ability for Java programmers to maintain both encapsulation and separation of concerns, two core object-oriented principles. An inner type's type-ancestry remains completely separate from that of the outer type, but has access to any private fields and methods declared within its outer type. This makes it possible for the inner type to implement other interfaces and/or extend a different parent class than the outer type. The outer type can separate its function and scope from that of the inner type, while still sharing data with the type that otherwise would be inconvenient or insecure. It also provides a convenient way to keep related data and functions together in one file while exploiting the type system for such features as polymorphism and inheritance.

Since inner types are fundamentally no different than outer types, BBj® has provided the ability to call methods of inner classes for as long as it has supported objects. On the other hand, instantiating an inner class or otherwise referring to an inner type was only available through the Java Reflection API. This also meant that using the DECLARE verb to associate an inner type with a variable was illegal in BBj.

How They Work

In 8.10 and higher, BBj supports the use of inner classes anywhere previously allowed a top-level type. USE statements, DECLARE statements, static method invocations and the "new" operator all accept Java's inner types, making inner types more accessible to BBj programmers.

The syntax for referencing an inner type is similar to referencing a static field or method of a top-level Java class. Simply separate the outer type from the inner type with a dot (.). An example of this is the inner interface "Entry" of the interface "java.util.Map." A program may refer to the "Entry" interface in the following three ways:

1. Explicitly refer to the full type name "java.util.Map.Entry."

2. Include the statement
   ```
   USE java.util.Map
   ```
   and refer to "Map.Entry" elsewhere in the program

3. Include the statement
   ```
   USE java.util.Map.Entry
   ```
   and refer to the interface "Entry" in the program.

continued...

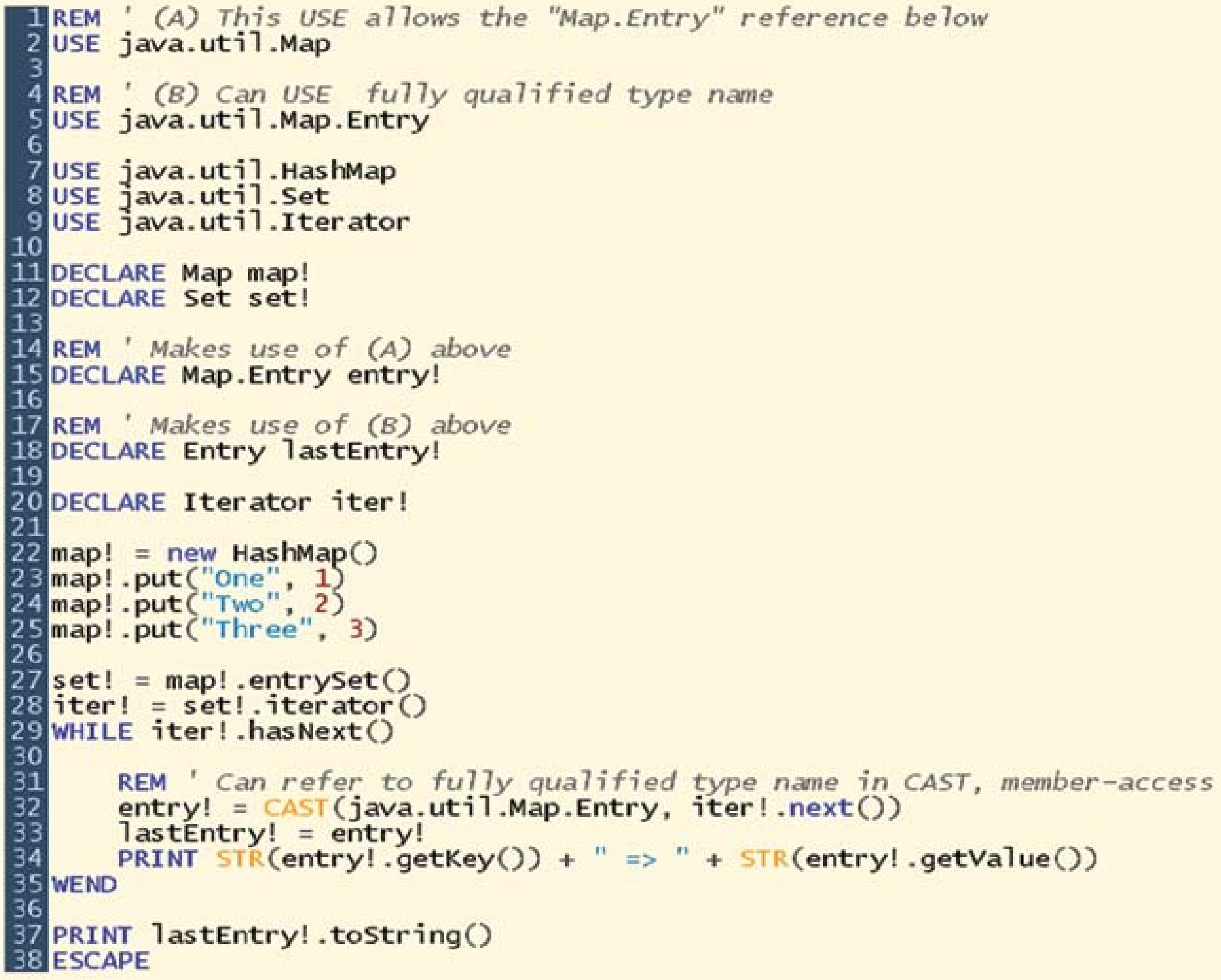
Adam Hawthorne
Software Engineer

Here are two examples of the use of inner classes and the code that produced each one.

Example 1

The code shown in **Figure 1**, illustrates several uses of java.util.Map and its inner type Entry. The Map.entrySet() method returns a java.util. Set containing objects implementing the Map.Entry interface. Iterating over the entire Set of mappings from key to value using only variables associated with a type using the DECLARE verb was not possible. Any program that used the Set returned by Map.entrySet() would not cleanly type check with BASIS' compiler, BBjCpl.

The program `InnerClass-1.src`, shown in **Figure 1,** may now DECLARE its variables like this:



```
REM ' (A) This USE allows the "Map.Entry" reference below
USE java.util.Map

REM ' (B) Can USE  fully qualified type name
USE java.util.Map.Entry

USE java.util.HashMap
USE java.util.Set
USE java.util.Iterator

DECLARE Map map!
DECLARE Set set!

REM ' Makes use of (A) above
DECLARE Map.Entry entry!

REM ' Makes use of (B) above
DECLARE Entry lastEntry!

DECLARE Iterator iter!

map! = new HashMap()
map!.put("One", 1)
map!.put("Two", 2)
map!.put("Three", 3)

set! = map!.entrySet()
iter! = set!.iterator()
WHILE iter!.hasNext()

    REM ' Can refer to fully qualified type name in CAST, member-access
    entry! = CAST(java.util.Map.Entry, iter!.next())
    lastEntry! = entry!
    PRINT STR(entry!.getKey()) + " => " + STR(entry!.getValue())
WEND

PRINT lastEntry!.toString()
ESCAPE
```

Figure 1. `InnerClass-1.src` showing several uses of java.util.Map

continued...

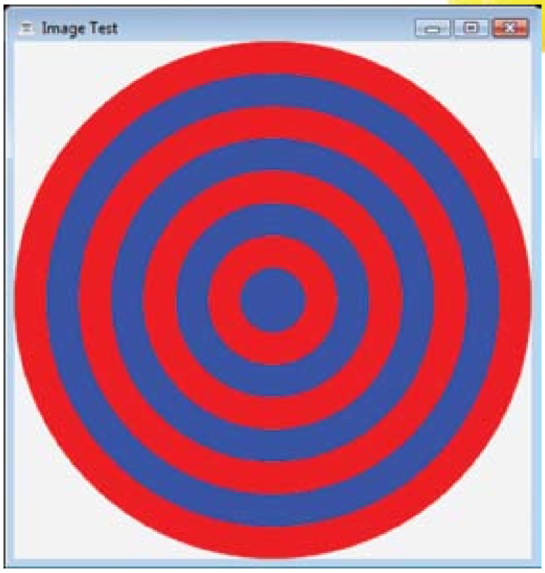


Figure 2. Result of `InnerClass-2.src`

Example 2

The program named `InnerClass-2.src` available for download. Here is a more involved example that includes constructing inner classes included in the Java 2D API. This API allows drawing geometric shapes to an image that may be used later on a BBjWrappedJComponent. The result of running `InnerClass-2.src` appears in **Figure 2**.

Summary

Although **Figure 2** may be the most exciting thing in this article, the ability for developers to use inner types will continue to reduce programming errors and simplify programming in BBj. Reducing errors and simplifying…a programmers maxim, and always a cause for celebration.

Download the code samples from
www.basis.com/advantage/mag-v12n1/InnerTypes.zip

Real-world BASIS Example

By Brian Hipple

When developing the demos for the latest TechCon, we ran into many issues that were due to the lack of support of inner types in BBj. We found out that we could not declare variables that were inner types, which led to code completion not working in the BASIS IDE as well as receiving type check warnings when compiling. We were able to create inner types by incorporating a fairly complicated BBj utility class (InnerClassFactory) that used Java reflection. The JFree chart package, which many of the demos utilized, uses inner types for specifying the dial pointer on a chart. **Figure 1** shows the creation of a client side dial pointer; first the old complicated way and then the new simple way with inner type support.

```
REM Create a client-side dial pointer using the InnerClassFactory
factory! = new InnerClassFactory("org.jfree.experimental.chart.plot.dial.DialPointer$Pointer", 1)
factory!.addArgument("int", new java.lang.Integer(1))
pointer! = CAST(DialPointer@, factory!.newInstance())

REM Create a client-side dial pointer using InnerClasses
pointer! = new DialPointer.Pin@(1)
```

Figure 1. Create a client-side dial pointer